AUTOMATED FUNCTIONAL TESTING

Recipes

Behaviour-Driven Acceptance Testing

A comprehensive guide with 700+ recipes for building reliable, maintainable, and scalable automated test suites using Behaviour-Driven Development.

Rajab Natshah

with Tasneem Natshah & Shaimaa Natshah

Table of Contents

- Dedication and Acknowledgments

Part I: Foundations

- Chapter 1: The World of BDD
- Chapter 2: Setting Up webship-js
- Chapter 3: The Gherkin Deep Dive
- Chapter 4: Your First Real Test Suite

Part II: Browser Testing Recipes

- Chapter 5: Navigation Recipes
- Chapter 6: Form Testing Recipes
- Chapter 7: Waiting and Dynamic Content
- Chapter 8: Modal and Dialog Testing
- Chapter 9: Table and List Testing
- Chapter 10: Authentication and Authorization Testing

Part III: API and Advanced Testing

- Chapter 11: API Testing Recipes
- Chapter 12: Multi-Browser Testing
- Chapter 13: Responsive and Mobile Testing
- Chapter 14: Screenshot and Visual Testing
- Chapter 15: Accessibility Testing

Part IV: Architecture and Organization

- Chapter 16: The PlaywrightWorld Deep Dive
- Chapter 17: Writing Custom Step Definitions
- Chapter 18: Hooks and Lifecycle
- Chapter 19: Organizing a Large Test Suite
- Chapter 20: Design Patterns for Tests

Part V: Troubleshooting and Maintenance

- Chapter 21: When Tests Go Bad
- Chapter 22: Debugging Techniques
- Chapter 23: Test Maintenance
- Chapter 24: Performance Optimization

Part VI: Real World

- Chapter 25: CI/CD Integration
- Chapter 26: Testing CMS Applications
- Chapter 27: Testing Single-Page Applications
- Chapter 28: Building a Complete Test Suite
- Chapter 29: BDD in the Age of AI
- Chapter 30: Best Practices and Pro Tips

Part VII: The AI-Powered Testing Ecosystem

- Chapter 31: The Webship-js AI Agent Ecosystem
- Chapter 32: Real-World Testing, International and Enterprise Websites
- Chapter 33: The Complete Step Definition Reference with Examples

Part VIII: Expanded Ecosystem

- Chapter 34: Advanced Selectors, Named Locators, Spatial Relations, Viewports

Part IX: Structured Prompt-Driven Development

- Chapter 35: Structured Prompt-Driven Development with Webship-js

Part X: Modern Test Authoring

- Chapter 36: Web-First Assertions
- Chapter 37: Iframe Testing
- Chapter 38: Clock Mocking and Time Travel
- Chapter 39: Network Mocking and Request Recording
- Chapter 40: File Downloads and Video Recording
- Chapter 41: XML and YAML Response Testing
- Chapter 42: Pointer Actions, Drag and Drop, Keyboard
- Chapter 43: Advanced Field Types

Part XI: Production Practice

- Chapter 44: JavaScript Errors and Tester-Friendly Error Handling
- Chapter 45: CI/CD Provider Deep-Dive
- Chapter 46: Video Recording Deep-Dive
- Chapter 47: Lessons From Varbase
- Chapter 48: Storage, Cookies, and Selectors, Deep Dive
- Chapter 49: Reference Patterns Library
- Chapter 50: Patterns vs Anti-Patterns
- Chapter 51: Foundations, Wisdom From the TDD and BDD Canon

Appendices

- Appendix A: Why Playwright Won, The Journey from Nightwatch.js
- Appendix B: webship-js Step Definition Quick Reference
- Appendix D: Further Reading
- About the Authors

Dedication and Acknowledgments

Dedication

Our Father

The first teacher.

This book carries his fingerprints, even where his name does not appear in the code. He taught us, before we ever met a compiler, that **work is a form of honesty**: that a half-done job is a debt to whoever finds it next, and that **a thing worth doing is worth doing in a way you would not be ashamed to sign**.

To our Father, for the values that shape every line of work.

Acknowledgments

Mohammed Razem

Founder and CEO of Vardot.

Mohammed has been part of the Drupal community since 2007, back in the Drupal 5 era, and in 2011 he founded **Vardot**, which he has grown into one of the largest Drupal-based digital-solutions providers, a team of more than seventy consultants serving enterprises, governments, and non-profits. Vardot's enterprise distribution, **Varbase**, became the most-installed Drupal 9 and 10 distribution in the world, and its acceptance suite is the proving ground where much of webship-js was first stress-tested.

Rajab works with Mohammed at Vardot, and it was Mohammed who made room for an automated-testing product to be built inside the company: who signed the time, backed the engineering, and endorsed the public release of the tooling under permissive licences. He has spent close to two decades arguing, by example, that an agency does best when it gives back to the open-source projects it depends on.

The shorthand is simple: webship-js exists because Rajab believed in it and **because Mohammed Razem made the room to build it**.

- LinkedIn: linkedin.com/in/razem
- Vardot: vardot.com

To Mohammed, for the room and the mentorship that turned a side-project into a published tool.

The Open-Source Community

A book like this is a thin layer on top of decades of free work by other people, the maintainers of **Playwright**, **Cucumber**, and **Node.js**, and the writers, from **Martin Fowler** onward, whose ideas shaped how the field thinks about tests. The tools themselves are credited with links in Appendix D.

To every contributor, tester, writer, and maintainer whose work makes this book possible, thank you.

The names listed in these pages are the ones who wrote the words. The credit belongs to a community much larger than any one byline.

Part I: Foundations

Chapter 1: The World of BDD

1.1 The Problem with Traditional Testing

Let me paint a picture you have probably lived through.

A product manager writes a specification. It is 40 pages long, written in corporate language, and includes phrases like “the system shall provide seamless integration with downstream consumer-facing touchpoints.” Nobody reads the whole thing. The developer skims it, makes assumptions, and starts building.

Three weeks later, the developer ships the feature. QA gets it. They find 14 bugs. The developer says “that is not a bug, it is working as designed.” The PM says “that is not what I asked for.” QA says “it crashes on Firefox.” Everyone is frustrated. The deadline has passed.

This is not a failure of talent. Everyone on this team is capable. It is a failure of **communication**. The specification was ambiguous. The developer interpreted it one way. QA interpreted it another. The PM had a third mental model. Nobody realized they were talking about different things until the code was already written.

Now multiply this by every feature, every sprint, every team. The cost is staggering. Studies show that bugs found in production cost **30x more** to fix than bugs caught during requirements gathering. Not 30%. Thirty times.

The testing world tried to solve this with more testing. More unit tests. More integration tests. More manual QA cycles. More regression suites that take 8 hours to run. And it helped, to a point. But it never fixed the root cause: the gap between what someone wanted and what someone built.

And if you are the QA lead who owns testing across ten or twenty projects, not one, the gap compounds. Each project has its own PM with their own vocabulary, its own developers with their own assumptions, and its own pile of test cases written in whatever style the last contractor preferred. The Tuesday before a release, someone asks you “is project X safe to ship?” and the honest answer is “I can tell you which tests passed, but I cannot tell you what they actually prove.” That is the problem this book keeps coming back to: one team, many projects, and no shared definition of “correct.” I lived this while maintaining the test suites for the Varbase distribution at Vardot, where the same features had to be verified across dozens of sites. The fix was never more tests. It was a shared language for what the tests mean.

The Waterfall Testing Model

In the traditional waterfall model, testing happens at the end:

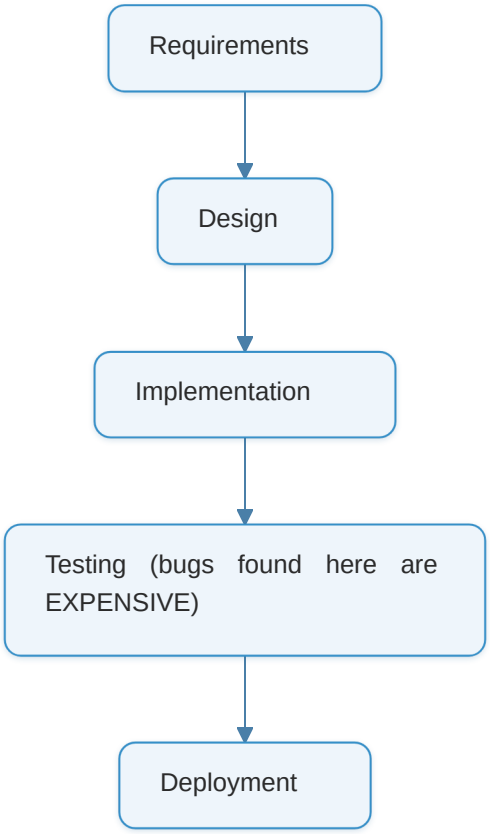


FIGURE 1.1 *The Waterfall Testing Model*

The problem is obvious: testing happens too late. By the time QA finds that the login page does not handle Unicode characters, the backend has been built, the frontend has been styled, the API has been deployed, and changing it requires touching five different systems.

The Agile Shift

Agile methodologies moved testing earlier. Instead of one big test phase, you test in every sprint. But the fundamental problem remained: **who decides what "correct" means?** The developer writes tests based on their understanding. QA writes test cases based on their understanding. Neither might match what the product owner actually wanted.

What if there was a way to define "correct" before anyone writes a single line of code? What if that definition was so clear that a developer, a tester, and a product manager could all look at it and agree: "yes, that is exactly what we want"?

That is BDD.

1.2 A Brief History: From TDD to BDD

Kent Beck and the Testing Revolution

In the late 1990s, Kent Beck was working on the Chrysler Comprehensive Compensation System (C3) project. He was practicing what would become known as Extreme Programming (XP), and at its core was an idea that seemed radical at the time: **write the test before you write the code.**

Test-Driven Development (TDD) was born from a simple observation. When you write tests first:

1. You think about what the code should do before you think about how
2. You end up with better-designed code (because testable code is well-structured code)
3. You have a regression safety net from day one
4. You know exactly when you are done (all tests pass)

TDD follows a tight cycle:

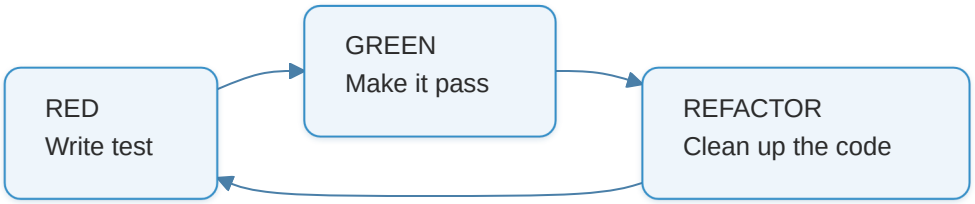


FIGURE 1.2 *Kent Beck and the Testing Revolution*

RED: Write a test that fails. This proves the test is actually testing something. **GREEN**: Write the minimum code to make the test pass. No more. **REFACTOR**: Clean up the code without changing behavior. The tests prove it still works.

TDD was a revolution. Teams that adopted it saw dramatic improvements in code quality. Kent Beck articulated four criteria for simple design:

1. **Passes all the tests** (correctness)
2. **Reveals all the intention** (readability)
3. **Contains no duplication** (maintainability)
4. **Uses fewest classes and methods** (simplicity)

But TDD had a limitation. It was a developer-centric practice. Tests were written in programming languages. They tested functions, classes, and methods. A product manager could not look at a JUnit test and understand what feature it was verifying. Tests were organized around technical components, not business features.

Dan North's Revelation

In the mid-2000s, Dan North was teaching TDD to developers and kept running into the same questions:

- "Where do I start?"
- "What do I test?"
- "What do I NOT test?"
- "How much testing is enough?"

North realized that the problem was not with TDD itself but with how developers thought about tests. They were writing tests for classes and methods. They should have been writing tests for **behaviors**.

He described BDD as:

> *"A second-generation, outside-in, pull-based, multiple-stakeholder, multiple-scale, high-automation, agile methodology."*

Let me unpack that:

- **Second-generation**: Built on TDD's foundation
- **Outside-in**: Start from the user's perspective, work inward
- **Pull-based**: Features are pulled by business need, not pushed by developer curiosity
- **Multiple-stakeholder**: Everyone participates, not just developers
- **Multiple-scale**: Works for units, features, and systems
- **High-automation**: Everything that can be automated, is automated

But you do not need to memorize that definition. Here is the practical version:

BDD is writing executable specifications in a language everyone on the team understands.

The Evolution Timeline

1999 — `Kent Beck publishes "Extreme Programming Explained"`
TDD becomes a defined practice

2003 — Dan North starts exploring "behavior" as an alternative to "test" in naming and thinking

2006 — `Dan North publishes "Introducing BDD"`
JBehave created (first BDD tool, Java)

2008 — Aslak Hellesøy creates Cucumber (Ruby)
Gherkin syntax introduced

2011 — `SpecFlow brings Cucumber to .NET`
Cucumber-JVM brings it to Java

2012 — Cucumber official release (March 2012)
Gherkin translated to 37+ languages

2015 — `Cucumber-js brings BDD to JavaScript/Node.js`

2020 — Playwright released (Microsoft)
Modern browser automation

2024 — `webship-js 2.0 combines Playwright + Cucumber-js`
Pre-built steps for instant productivity

2026 — AI-assisted BDD becomes mainstream
Test-Drive-Develop framework

FIGURE 1.3 *The Evolution Timeline*

1.3 What Is Behaviour-Driven Development?

At its core, BDD is a collaboration methodology. Yes, it involves tools and tests, but the real value is in the **conversations** that happen while writing features.

Here is what the Cucumber Book says, and I agree completely: “Cucumber is really a **collaboration tool**, not just a testing tool.”

Let me explain with an example.

A Real Conversation

Imagine a Three Amigos session for an e-commerce checkout feature:

Product Owner: “When the user checks out, we charge their credit card and send a confirmation email.”

Developer: “What happens if the credit card is declined?”

Product Owner: “Show an error message.”

Tester: “What if the card is charged but the email server is down? Do we still confirm the order?”

Product Owner: “…yes. The order is confirmed even if the email fails. We will retry the email.”

Tester: “What about partial failures? The card is charged $50 but the item costs $45 after a coupon is applied?”

Product Owner: “Wait, where does the coupon get applied? Before or after tax?”

Developer: “That depends on the state. Some states require tax on the pre-coupon amount.”

That five-minute conversation just prevented three bugs, clarified a business rule, and identified a tax compliance requirement that would have cost thousands to fix in production.

Now let us write the scenarios:

```
Feature: Checkout
  As a customer with items in my cart
  I want to complete my purchase
  So that I receive my products

  Scenario: Successful checkout with valid credit card
    Given I have items totaling "$45.00" in my cart
    And I am on the checkout page
    When I fill in my credit card information
    And I press "Complete Purchase"
    And wait for AJAX to finish
    Then I should see "Order Confirmed"
    And I should see "A confirmation email has been sent"

  Scenario: Checkout with declined credit card
    Given I have items totaling "$45.00" in my cart
    And I am on the checkout page
    When I fill in an invalid credit card
    And I press "Complete Purchase"
    And wait for AJAX to finish
    Then I should see "Your card was declined"
    And I should not see "Order Confirmed"

  Scenario: Order confirmed even when email fails
    Given I have items totaling "$45.00" in my cart
    And I am on the checkout page
    When I fill in my credit card information
    And I press "Complete Purchase"
    And wait for AJAX to finish
    Then I should see "Order Confirmed"
```

Every person in that meeting can read these scenarios. Every person agrees on what “correct” looks like. And now these scenarios are **executable**, they run against the real application and verify it works.

A note on that `wait for AJAX to finish` line, because it will appear throughout this book: it is a real webship-js step, and under the hood it calls smartSettle, a routine that watches the page's live counters (`window.__webshipAjaxCount` for in-flight fetch/XHR calls, `window.__webshipPendingTimers` for pending timeouts, and a MutationObserver timestamp for DOM changes) and returns only when the page has genuinely gone quiet. The point for this chapter is simpler: there is no `sleep 5` anywhere. The scenario says what the user waits for, not how many seconds to guess. That discipline, wait for events, never for time, is what separates a suite people trust from a suite people rerun until it passes.

At scale: Here is what the product owner gets out of this. When these three scenarios pass on staging, the PO does not need to take engineering's word that checkout works, they can read the green report themselves, line by line, in the same words they used in the meeting. Across many projects, that turns sign-off from a trust exercise into a reading exercise. The acceptance criteria the PO wrote ARE the tests; QA never rewrites them from scratch in a separate tool.

The Formal Definition

BDD combines three established practices:

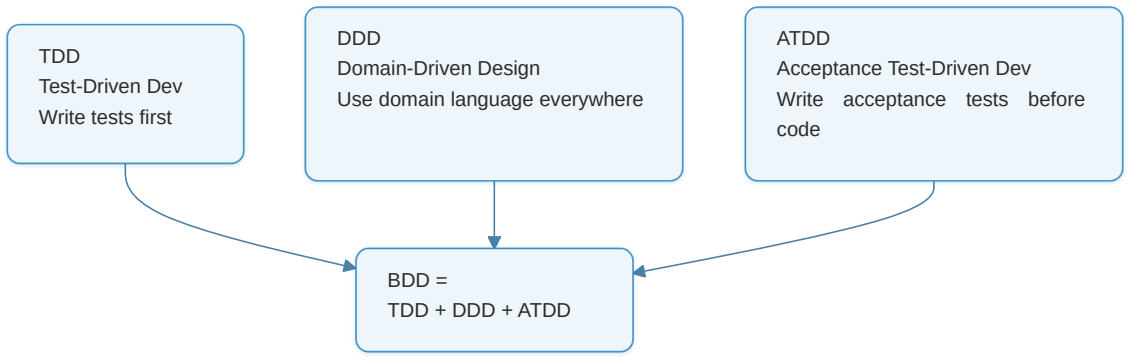


FIGURE 1.4 *The Formal Definition*

- **TDD** gives us the test-first discipline
- **DDD** gives us the shared language (ubiquitous language)

- **ATDD** gives us the outside-in perspective (start from user acceptance)

KISS and YAGNI

BDD embraces two principles from agile development:

KISS (Keep It Simple, Stupid): Write the simplest test that could fail. Write the simplest code that makes it pass. Do not over-engineer.

YAGNI (You Ain't Gonna Need It): Do not write tests for features nobody asked for. Do not add functionality "just in case." Every scenario should trace back to a real business need.

Minimum Marketable Feature (MMF)

BDD focuses on the **Minimum Marketable Feature**, the smallest piece of functionality that delivers value to the user. Instead of planning a 6-month release with 200 features, BDD teams deliver a single feature that works perfectly, then add the next one.

Each MMF gets its own feature file with scenarios. When all scenarios pass, the feature is done. Ship it.

1.4 BDD vs TDD: Features vs Components

This distinction is critical. Let me make it concrete.

What TDD Tests

TDD tests **components**, classes, functions, methods. A TDD test for a calculator might look like:

```
// TDD: Testing a component
describe('Calculator', () => {
  it('should add two numbers', () => {
    const calc = new Calculator();
    calc.push(50);
    calc.push(70);
    expect(calc.add()).toBe(120);
  });

  it('should throw on division by zero', () => {
    const calc = new Calculator();
    calc.push(10);
    calc.push(0);
    expect(() => calc.divide()).toThrow('Division by zero');
  });
});
```

If this test fails, you know the **Calculator class** is broken. But you do not know which **feature** is affected. Maybe the calculator is used in the checkout flow, the reporting dashboard, and the admin panel. Which user-facing feature broke?

What BDD Tests

BDD tests **features**, user actions and business behaviors:

```
Feature: Addition
  In order to avoid silly mistakes
  As a math idiot
  I want to be told the sum of two numbers

  Scenario: Add two numbers
    Given I have entered 50 into the calculator
    And I have entered 70 into the calculator
    When I press add
    Then the result should be 120 on the screen
```

If this test fails, you know the **addition feature** is broken. Any user who needs to add numbers is affected. The business impact is clear immediately.

Comparison Table

	TDD	BDD
Tests what?	Components	Features
Written by?	Developers	Team (3 Amigos)
Language?	Programming language	Gherkin (plain text)
Audience?	Developers	Everyone
Granularity?	Function/class	User story/feature
On failure?	Know which component	Know which feature
Maintenance?	Devs maintain	Team maintains
Documentation?	Code comments	Living specification
Speed?	Very fast (ms)	Slower (seconds)
When to run?	Every save	Every commit/push

When to Use Each

- **Small project, one developer**: TDD alone might suffice
- **Larger project, multiple stakeholders**: BDD for features, TDD for components
- **API-heavy project**: BDD for endpoints, TDD for business logic
- **Frontend-heavy project**: BDD for user flows, TDD for utilities

For most real-world projects, **combine both**. BDD at the feature level. TDD at the component level. They complement each other perfectly.

At scale: The “on failure” row in that table matters more than it looks. When you run nightly suites across 20 projects, the morning triage question is never “which class broke?”, it is “which user-facing feature broke, on which product, and does it block the release?” Feature-level failures answer that directly: `login.feature` failed on project X means login is broken on project X, and the delivery manager can read that sentence without an engineer translating. Component-level failures need a developer from that specific project to interpret them, and across 20 projects you do not have 20 spare developers at 9 a.m.

1.5 The Three Amigos Pattern

The Three Amigos is not a meeting. It is a **conversation format**. Three perspectives come together before a single line of code is written:

The Three Perspectives

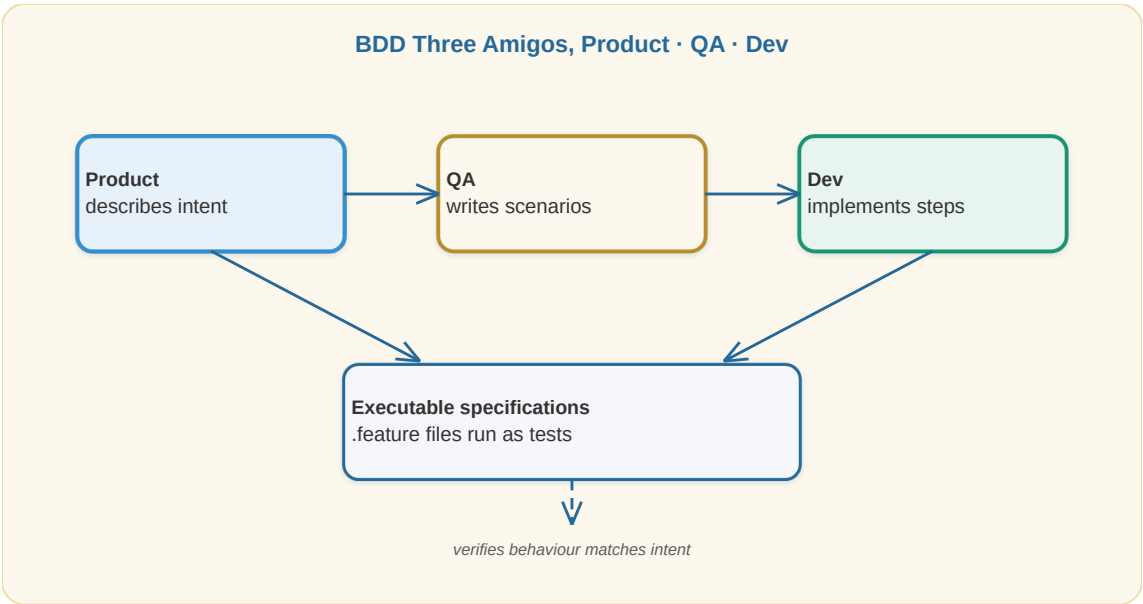


FIGURE 1.5 *The Three Perspectives*

Running a Three Amigos Session

Duration: 15-30 minutes per user story. No longer.

Input: A user story or requirement. Even a one-liner is fine.

Process:

1. Product Owner describes the feature in their words
2. Developer asks “how” questions (technical constraints)
3. Tester asks “what if” questions (edge cases)
4. Together, they write the scenarios in Gherkin
5. Everyone agrees: “if these scenarios pass, this feature is done”

Output: A set of Gherkin scenarios that everyone understands and agrees on.

Example: Three Amigos for a Search Feature

PM: “Users should be able to search for products.”

Dev: “Full-text search or just exact match? Do we search titles only or descriptions too?”

PM: “Start with titles. We can add descriptions later.”

Tester: “What happens when there are no results?”

PM: “Show a helpful message. Maybe suggest similar products.”

Tester: “What about special characters? SQL injection?”

Dev: “We parameterize all queries, but good point, let us test that it does not crash.”

Tester: “What about very long search terms?”

PM: “Truncate to 200 characters.”

Result:

```
Feature: Product Search
  As a customer
  I want to search for products by title
  So that I can find what I am looking for

  Scenario: Search with matching results
    Given I am on homepage
    When I fill in "Search" with "laptop"
    And I press "Search"
    And wait for AJAX to finish
    Then I should see "Search results for: laptop"
    And I should see "MacBook Pro"

  Scenario: Search with no results
    Given I am on homepage
    When I fill in "Search" with "xyznonexistent"
    And I press "Search"
    And wait for AJAX to finish
    Then I should see "No products found"
    And I should see "Try a different search term"

  Scenario: Search with special characters
    Given I am on homepage
    When I fill in "Search" with "laptop'; DROP TABLE products;--"
    And I press "Search"
    And wait for AJAX to finish
    Then I should see "No products found"
    And I should not see "error"
```

Notice: the security test is right there alongside the happy path. The tester's perspective caught it. The PM agreed it mattered. The developer knows to parameterize queries. All from a 15-minute conversation.

At scale: When one QA team supports many product teams, you cannot sit in every Three Amigos session, and you should not try. What works instead: run the session with the first two or three features of each new project, write those scenarios together, and leave them in the repository as the house style. The next PO who writes acceptance criteria copies the shape of what is already there.

The scenarios become the onboarding document, the writing template, and the quality bar, all at once. The third project you onboard goes faster than the first because the conversation format travels with the files.

1.6 Given-When-Then: The Universal Language

The Given-When-Then format was inspired by the Arrange-Act-Assert pattern from TDD:

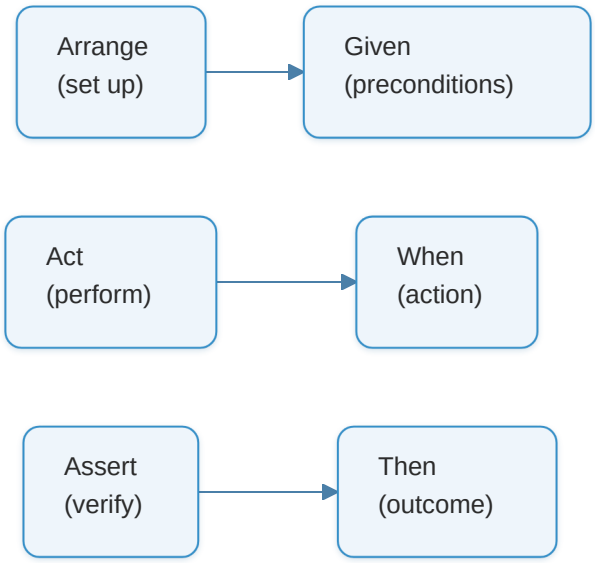


FIGURE 1.6 *Given-When-Then: The Universal Language*

But there is a critical difference. Arrange-Act-Assert is for developers. Given-When-Then is for everyone.

The Keywords in Detail

Given, Set up the world. Put the system in a known state.

```
Given I am on homepage
Given I am a logged in user with the "admin" user
Given I am on "/products"
```

Givens are preconditions. They describe the state **before** the interesting thing happens. They should not trigger the behavior being tested, they just set the stage.

When, The action. The thing the user does.

```
When I fill in "Email" with "user@example.com"
When I press "Submit"
When I click "Delete"
When I go to "/dashboard"
```

Each scenario should have one key **When** action. The When is the trigger, the event that causes the behavior being tested. You might have multiple When steps to describe a sequence of actions, but they should all serve the same behavior.

Then, The assertion. What should be true after the action.

```
Then I should see "Welcome back"
Then I should not see "Error"
Then the API response code should be 200
Then ".cart-badge" should be visible
```

Then steps verify the outcome. They check the observable result. Importantly, they check what the **user** can see, not what the database contains or what internal state changed.

One detail worth knowing early: in webship-js, Then steps are not one-shot checks. The web-first assertions (like `should be visible`) poll the live page every 100ms for up to 5 seconds before giving up. The assertion IS the wait. That single design decision eliminates the largest source of flaky tests, asserting a millisecond before the page finished rendering.

And / But, Continue the previous keyword type.

```
Given I am on homepage
  And I am a logged in user with the "admin" user   # This is a Given

When I fill in "Title" with "My Post"
  And I press "Publish"                              # This is a When

Then I should see "Post published"
  And I should not see "Draft"                       # This is a Then
  But I should not see "Error"                       # Also a Then
```

`And` and `But` are syntactic sugar. Cucumber treats them identically to whatever keyword came before. They exist purely for readability.

The Golden Rules

1. **Given steps should not trigger the behavior being tested.** If your test is about login, the Given should set up the preconditions (navigate to login page), not perform the login itself.
2. **When steps should describe user actions, not system actions.** “When the system processes the payment” is wrong. “When I press ‘Pay Now’” is right.
3. **Then steps should check observable outcomes.** “Then the database should have a new user record” is wrong. “Then I should see ‘Registration successful’” is right.
4. **Keep each scenario focused on one behavior.** If you are testing login AND dashboard display AND notification count, split them into three scenarios.

1.7 Living Documentation

This is one of the most undervalued benefits of BDD.

The Documentation Problem

Every software project has documentation. And every software project has documentation that is wrong.

Requirements documents become outdated the moment development starts. Wiki pages describe features that were redesigned months ago. API docs reference endpoints that no longer exist. README files have installation instructions that have not worked since 2023.

The root cause is simple: documentation and code are maintained separately. They drift apart over time. Nobody notices until someone follows the docs and gets lost.

Features as Specification

BDD feature files solve this problem because they are **both** documentation and tests:

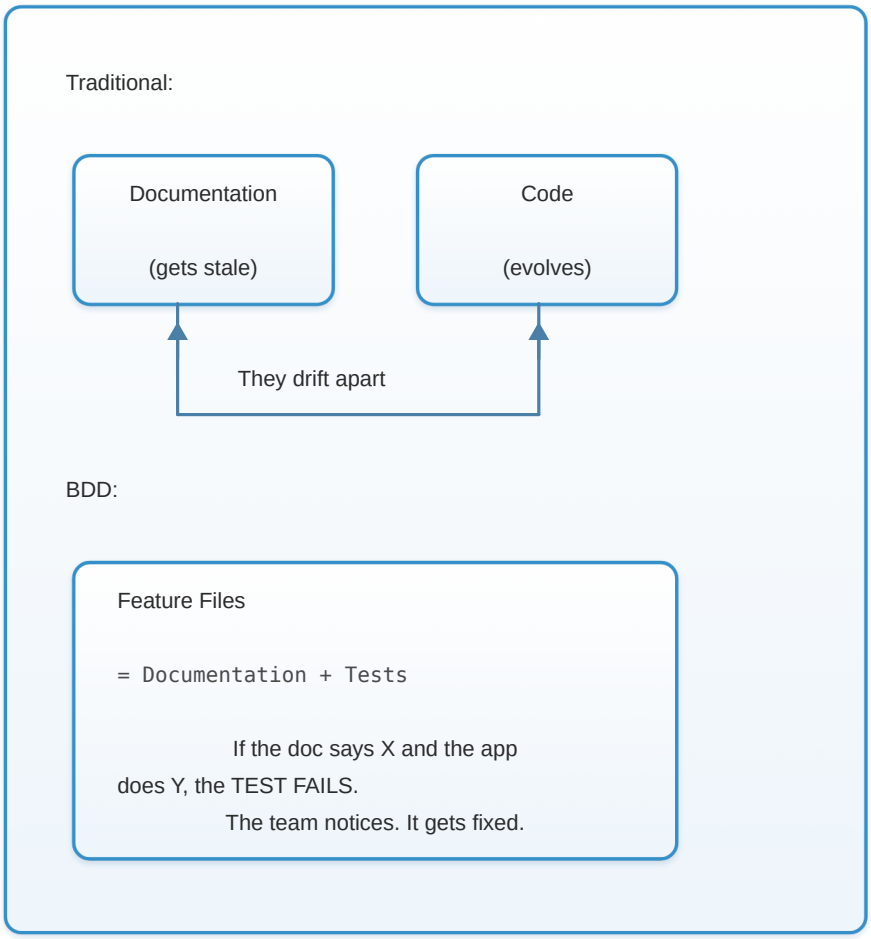


FIGURE 1.7 *Features as Specification*

When a feature file says:

```
Scenario: Successful login
  Given I am on homepage
  When I go to "/login"
  And I fill in "Email" with "user@example.com"
  And I fill in "Password" with "ValidPass123"
  And I press "Sign In"
  Then I should see "Dashboard"
```

If someone redesigns the login page and changes “Sign In” to “Log In”, the test fails. The team updates the feature file. The documentation stays current. Automatically. Without anyone having to “remember to update the docs.”

The Regression Safety Net

Living documentation gives you something else for free: **regression testing**.

Every feature file is a specification of how the system should behave. Every time you run the test suite, you verify that every specified behavior still works. If a code change breaks something, you know immediately, and you know exactly which feature is broken.

This is far more valuable than a test suite organized by class or module. When the `UserController.test.js` fails, you have to figure out which feature is affected. When `login.feature` fails, the feature that is broken is right there in the filename.

At scale: Living documentation pays out twice when you run many projects. First, for the delivery manager: a feature file that passed last night is evidence, not opinion, “is release X safe?” becomes “here are the 84 behaviors we verified at 2 a.m., read any of them.” Second, for the team itself: when a QA engineer rotates from project A to project B, the feature files are the spec for project B. Nobody has to schedule a knowledge-transfer meeting for documentation that executes itself nightly. The catch is consistency, this only works if all your projects write features in the same style, which is exactly why the conventions in Chapter 3 matter more than any individual scenario.

1.8 The BDD Workflow Cycle

BDD follows a cycle similar to TDD but at a higher level:

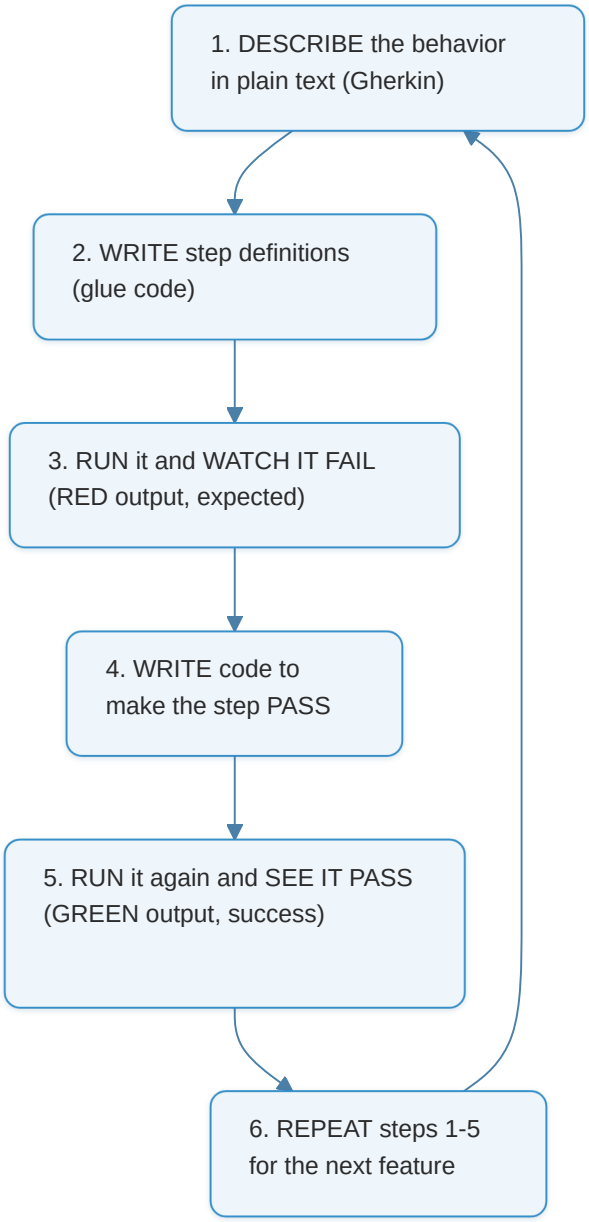


FIGURE 1.8 *The BDD Workflow Cycle*

“Repeat until green like a Cuke.”

Step by Step

1. **Describe the behavior in plain text.** Write a Gherkin feature file with one or more scenarios. This is the specification.
2. **Write step definitions.** Connect the plain-text steps to code. With webship-js, most steps are already written for you, 411 built-in steps across 36 step files cover navigation, forms, web-first assertions, API calls, cookies, accessibility audits, network mocking, and more. For a typical web project, you reach for custom step definitions only when your domain demands it.
3. **Run it and watch it fail.** The test should fail because the feature does not exist yet. This is the “RED” state. If it passes without any code, either the test is wrong or the feature already exists.
4. **Write code to make the step pass.** Implement the minimum code needed. Do not over-engineer. Do not add features nobody asked for.
5. **Run it again and see it pass.** The test should now be “GREEN.” If it is not, debug and fix.
6. **Repeat.** Pick up the next scenario or the next feature. Keep going “until the money runs out” (as the jf13 presentation humorously puts it).

The Cucumber Execution Flow

When Cucumber runs, each step results in one of four outcomes:

OUTCOME	COLOR	MEANING
Passed	**GREEN**	Step ran and assertion succeeded
Failed	**RED**	Step ran but assertion failed
Pending	YELLOW	Step definition exists but incomplete
Undefined	YELLOW	No step definition found
Skipped	CYAN	Step after a failure (not executed)

When a step fails, all subsequent steps in that scenario are **skipped**. Cucumber does not try to run steps after a failure because the system is in an unknown state.

One more thing happens between your steps that you never see in the output. After every state-changing step, webship-js runs an AfterStep hook that calls smartSettle with a 1500ms budget, it waits for in-flight AJAX to reach zero, pending timers to drain, and the DOM to stay quiet for at least 250ms before the next step runs. This is why a scenario that clicks, fills, and asserts in quick succession does not trip over its own speed. The repo rule behind it is blunt and worth adopting everywhere: no static sleeps, anywhere. Wait for events, not for time.

1.9 Cucumber: The Right Tool

The Tools Landscape

There are many BDD and testing tools available. They fall on a spectrum from “very technical” to “very accessible”:

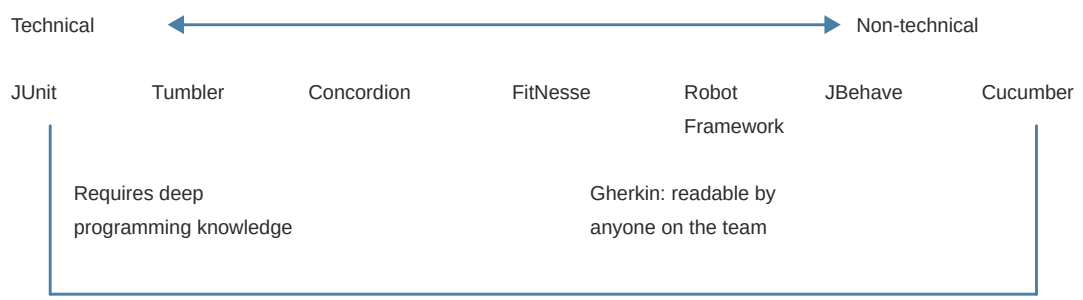


FIGURE 1.9 *The Tools Landscape*

Cucumber sits at the far right of this spectrum. It is the **least technical** BDD tool, which means it is the most accessible. Non-developers can read, write, and maintain Gherkin scenarios. That is its superpower.

Why Cucumber (and by extension, webship-js)?

1. **It descends from RSpec.** It has a mature, battle-tested lineage in the testing world.
2. **It is actively maintained.** Cucumber has been an active open-source project since 2008 with regular releases.
3. **It supports multiple languages.** Gherkin can be written in 47 human languages. Step definitions can be written in Ruby, Java, JavaScript, Python, .NET, and more.
4. **The audience is everyone.** Feature files are read by customers and developers. They are maintained by product owners and developers. They serve both as specification and as tests.

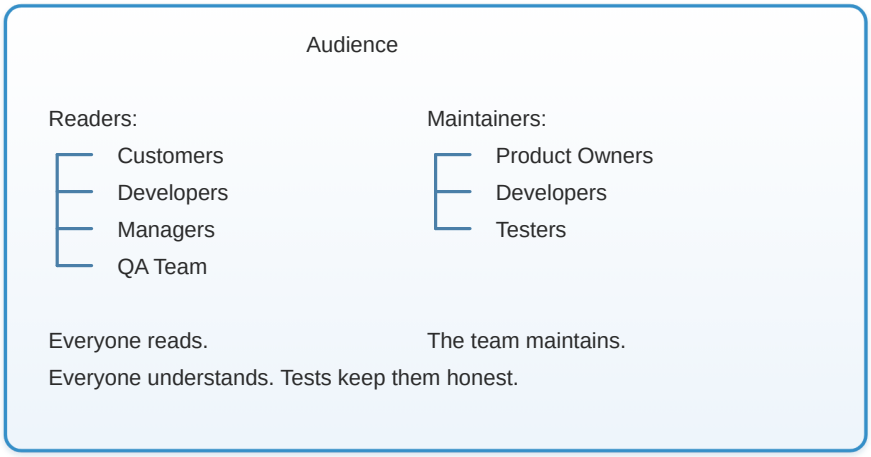


FIGURE 1.10 *Why Cucumber (and by extension, webship-js)?*

The Cucumber Architecture Pattern

Cucumber operates in three layers:

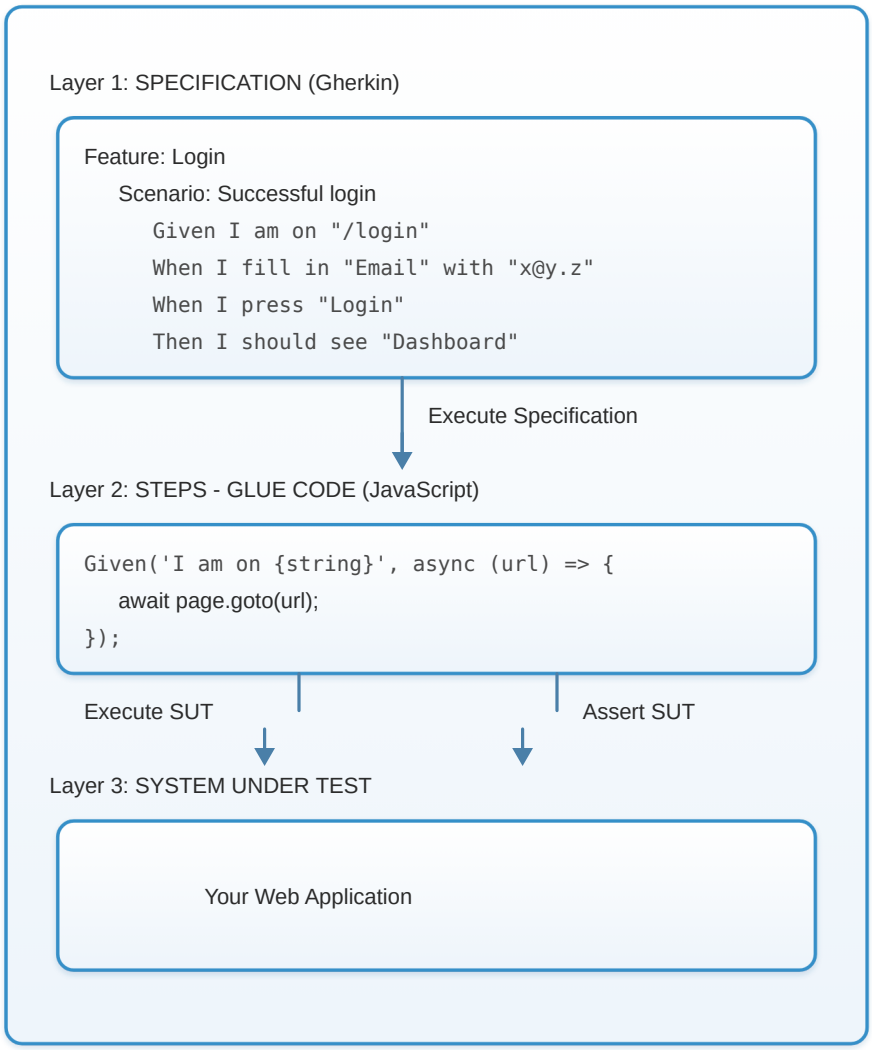


FIGURE 1.11 *The Cucumber Architecture Pattern*

The specification (Gherkin) drives the glue code (step definitions), which drives the system under test (your application). The step definitions both **execute** the SUT (navigate, click, fill) and **assert** against it (check content, verify state).

1.10 The Skills Pyramid

Building effective BDD tests requires skills from both the testing world and the software development world:

GHERKIN FEATURES	TESTING SKILL
	← Product Owners,
Step Definitions	Both Skills
	← Developers + Testers
Support Code	Software Design Skill
	← Developers
Automation Libs	Software Design Skill
	← Developers

Gherkin features are at the top. They require testing skill, knowing what to test, what edge cases matter, and how to express requirements clearly. Product owners and testers excel here.

Step definitions are in the middle. They bridge the human-readable Gherkin and the technical automation. Both testing and development skills are needed.

Support code and automation libraries are at the bottom. They require software design skill, clean architecture, good abstractions, maintainable code. Developers own this layer.

webship-js dramatically simplifies this pyramid by providing the middle and bottom layers out of the box. The 411 built-in step definitions are the middle layer, written once and battle-tested; Playwright 1.58 plus the PlaywrightWorld support code is the bottom. Your team only needs to focus on the top: writing great Gherkin features.

At scale: This pyramid is also a staffing model. If every project carries its own middle and bottom layers, every project needs a developer who maintains glue code, multiply that by 20 projects and you have a hidden engineering team doing nothing but step-definition upkeep. When the layers below Gherkin are shared (one framework version, one selector convention, one upgrade path), a QA engineer who learns the step vocabulary on one project is productive on all of them the same day. That is the real argument for a common framework across your portfolio: not elegance, headcount.

Chapter Summary

- Testing after development is expensive. BDD moves testing to the beginning.
- BDD = TDD + DDD. It combines test-first discipline with shared domain language.
- Given-When-Then maps to Arrange-Act-Assert but is readable by everyone.
- The Three Amigos (PO + Dev + Tester) collaborate on scenarios before coding.
- Feature files serve as living documentation that cannot go stale.
- The BDD cycle: Describe → Step Definition → Fail → Implement → Pass → Repeat.
- Cucumber is the most accessible BDD tool, readable by non-developers.
- webship-js provides the full BDD stack: Gherkin + Playwright + pre-built steps.

In the next chapter, we will set up webship-js and write our first real tests.

Chapter 2: Setting Up webship-js

2.1 What Is webship-js?

webship-js is an automated functional acceptance testing tool. Let me break down every word in that description because each one matters:

- **Automated**: Tests run themselves. No human clicks through the application.
- **Functional**: Tests verify that features work correctly. Not performance, not security, functional correctness.
- **Acceptance**: Tests are written from the user's perspective. They verify what the user accepts as "working."
- **Testing tool**: It is a framework, not a service. It runs on your machine or CI server.

Under the hood, webship-js combines three technologies:

1. **Cucumber-js**, the JavaScript implementation of Cucumber. Parses Gherkin feature files and matches steps to JavaScript functions.
2. **Playwright**, Microsoft's browser automation library (version 1.58 in webship-js 2.0). Controls Chromium, Firefox, and WebKit browsers. Handles navigation, clicking, typing, waiting, screenshots, and more.
3. **Pre-built step definitions**, 411 built-in steps organized into 36 step files, covering navigation, form interactions, web-first assertions, waiting, scrolling, modals and dialogs, cookies, accessibility audits, network mocking, video recording, and API testing. Each step file owns one domain (`form.steps.js`, `api.steps.js`, `a11y.steps.js`, and so on), which is also how you will find things when you want to read what a step really does.

Architecture Diagram

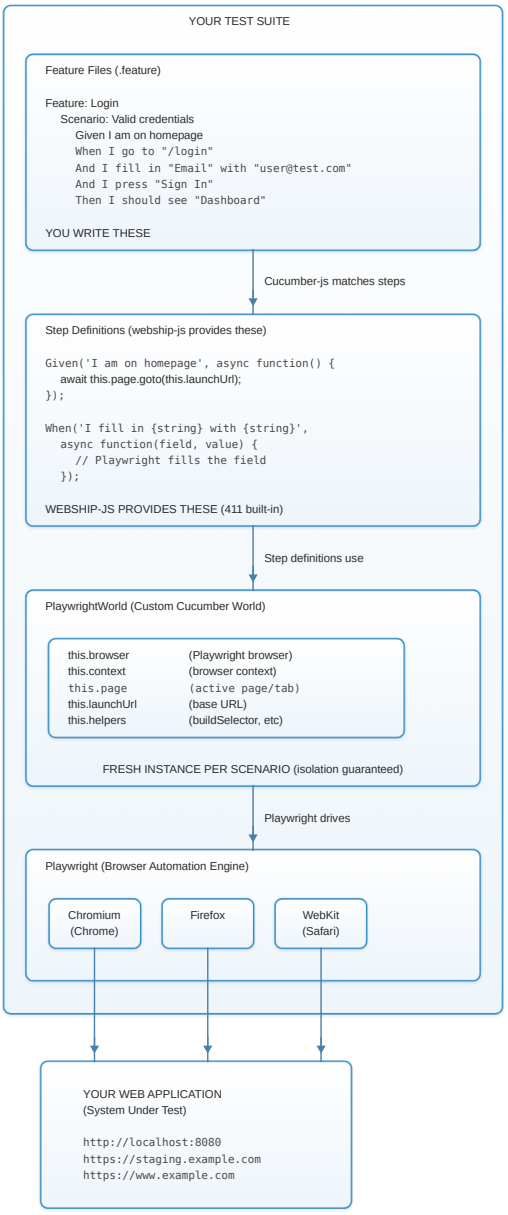


FIGURE 2.1 *Architecture Diagram*

How It Compares

You might be wondering how webship-js compares to other testing tools. Here is an honest comparison:

	WEBSHIP-JS	CYPRESS	SELENIUM	PLAYWRIGHT
			+ Cucumber	(raw)
Language	Gherkin +	JavaScript	Multiple	JavaScript
	JavaScript			
BDD Native?	Yes	No (addon)	Yes	No
Browser Support	Chromium	Chromium	All	Chromium
	Firefox	Firefox		Firefox
	WebKit	(limited)		WebKit
Pre-built Steps	411	None	None	None
Setup Time	Minutes	Minutes	Hours	Minutes
Custom JS Needed	Minimal	Everything	Moderate	Everything
Non-dev Friendly	Very	No	Somewhat	No
API Testing	Built-in	Built-in	Addon	Manual
Auto-wait	Yes	Yes	Manual	Yes

The key advantage of webship-js is **time to first test**. With Cypress or raw Playwright, you need to write all your test logic in JavaScript. With webship-js, you write Gherkin and the step definitions are already there.

At scale: Time to first test compounds across a portfolio. With a JavaScript-first tool, every new project starts with a developer building helpers, waits, and page objects before QA writes test one, and every project's helpers end up slightly different, so nothing transfers. With webship-js, the 411 steps are the same on project one and project forty. A QA engineer who can test your flagship product can open any other repo in your organization and be writing scenarios within the hour. That portability is the difference between a testing tool and a testing standard.

2.2 Prerequisites and Installation

System Requirements

- **Node.js**: Version 20 or higher (LTS recommended)
- **npm**: Version 9 or higher (comes with Node.js)
- **Operating System**: Linux, macOS, or Windows
- **Disk Space**: ~500MB (for Playwright browsers)
- **RAM**: 4GB minimum (8GB recommended for parallel testing)

Checking Your Environment

Open your terminal and verify:

```
# Check Node.js version
node --version
# Expected: v20.x.x or higher

# Check npm version
npm --version
# Expected: 9.x.x or higher
```

If Node.js is not installed or is too old, download the latest LTS from nodejs.org.

Setup 1: Setting Up a New webship-js Project from Scratch

Problem: A new product just landed on your team's plate and it has zero automated tests. Manual QA is keeping it alive, the developers "plan to add tests later," and you have been asked to fix that before the next release. You need a working test project today, not a framework evaluation, not a two-week tooling spike.

Solution:

```
# Step 1: Create the project directory
mkdir my-app-tests
cd my-app-tests

# Step 2: Initialize npm
npm init -y

# Step 3: Install webship-js
npm install webship-js --save-dev

# Step 4: Install Playwright browsers
npx playwright install

# Step 5: Create directory structure
mkdir -p tests/features
mkdir -p tests/step-definitions
mkdir -p tests/assets
```

How It Works:

`npm install webship-js --save-dev` installs:

- webship-js core (step definitions, PlaywrightWorld, helpers)
- @cucumber/cucumber (Gherkin parser, test runner)
- playwright (browser automation)
- axios (for API testing steps)

The `--save-dev` flag adds it to your devDependencies, which is correct because testing tools are not needed in production.

`npx playwright install` downloads the browser binaries:

- Chromium (Chrome-based browser)
- Firefox
- WebKit (Safari-based browser)

These are stored in your home directory's cache, not in the project. They are shared across all Playwright projects on your machine.

Note the Node.js requirement: webship-js 2.0 needs Node 20 or newer. If `npm install` succeeds but tests crash with syntax errors from inside `node_modules`, an old Node version is the first thing to check.

At scale: If this is the third or tenth project you are setting up, stop typing these commands by hand. Put the directory skeleton, a `cucumber.js` template, and the npm scripts into a starter repository (or a small `init` script) and clone it for every new product. The win is not the saved minutes, it is that every suite in your organization ends up with the same folder layout, so anyone can navigate any project's tests without asking where things live.

Setup 2: Adding webship-js to an Existing Project

Problem: The application already exists, it might be five years old, written in a framework nobody on the QA team codes in, and shipping every two weeks. You want acceptance tests around it without touching the application's own codebase or build, because the moment your test setup requires changes to the app, you need a developer's time and a release window just to get started.

Solution:

```
# Navigate to your existing project
cd /path/to/my-existing-project

# Install webship-js
npm install webship-js --save-dev

# Install browsers (if not already installed)
npx playwright install

# Create test directories
mkdir -p tests/features
mkdir -p tests/step-definitions
mkdir -p tests/assets
```

Add test scripts to your existing `package.json`:

```
{
  "name": "my-existing-project",
  "scripts": {
    "start": "node server.js",
    "test": "cucumber-js",
    "test:smoke": "cucumber-js --tags @smoke",
    "test:full": "cucumber-js --tags 'not @wip'"
  },
  "devDependencies": {
    "webship-js": "^2.0.0"
  }
}
```

How It Works:

webship-js lives alongside your application code. The test suite runs against your running application, it does not import your application code directly. It talks to the app the way a user does: over HTTP, through a real browser. This means webship-js tests work with ANY web application, regardless of the technology it is built with (React, Angular, PHP, Python, Java, etc.).

That black-box boundary is a feature, not a limitation. Because the tests never reach into application internals, they survive refactors, framework migrations, and even full rewrites, as long as the behavior stays the same, the suite stays green. I have watched teams replace an entire frontend stack underneath a passing acceptance suite. The suite was the proof the rewrite did not break anything users cared about.

At scale: Decide early whether tests live inside each application repo or in a separate tests-only repo per project, and apply that decision everywhere. In-repo means tests version together with the code and run on every PR; separate repos suit teams that test products they do not build (an agency QA team, a platform team testing vendor deployments). Both work. Ten projects with five doing it one way and five the other does not.

Setup 3: Installing Playwright Browsers

Problem: Tests pass on your laptop and crash in CI with `browserType.launch: Executable doesn't exist`, or worse, the browser launches and dies silently because a shared library is missing from the Docker image. It is the classic "works on my machine" of browser automation, and it usually shows up the first time someone wires the suite into a pipeline, on a Friday.

Solution:

```
# Install all browsers
npx playwright install

# Install only Chromium (fastest, sufficient for most testing)
npx playwright install chromium

# Install Chromium with system dependencies (needed for CI/Docker)
npx playwright install --with-deps chromium

# Install all browsers with system dependencies
npx playwright install --with-deps
```

How It Works:

Playwright bundles specific browser versions that are tested against its API. These are not your system browsers, they are standalone builds that Playwright controls directly.

Browser Storage Location:

```
~/.cache/ms-playwright/
├── chromium-XXXX/
│   └── chrome-linux/chrome
├── firefox-XXXX/
│   └── firefox/firefox
└── webkit-XXXX/
    └── pw_run.sh
```

FIGURE 2.2 *Setup 3: Installing Playwright Browsers*

The `--with-deps` flag installs system-level libraries (fonts, graphics libraries) that browsers need on headless Linux servers. Always use this flag in CI/CD environments.

Because the browser build is pinned to the Playwright version, the browser your test ran against locally is byte-for-byte the browser CI runs, if your Playwright versions match. That "if" is the catch. A lockfile drift between projects means project A tests on one Chromium build and project B on another, and one of them will eventually hit a rendering difference the other cannot reproduce.

At scale: This is a DevOps problem before it is a QA problem. Cache `~/.cache/ms-playwright/` in your CI configuration keyed on the Playwright version, downloading ~500MB of browsers for 40 nightly suites is real bandwidth and real minutes. Better still, build one base CI image with browsers and dependencies baked in, version it, and point every project's pipeline at it. When you upgrade Playwright, you upgrade the image once and every project moves together instead of drifting one lockfile at a time.

2.3 Project Structure

A well-organized webship-js project follows this structure:

```
my-app-tests/
├── cucumber.js                         # Cucumber configuration
├── package.json                        # Project dependencies
├── package-lock.json                   # Dependency lock file
│
├── tests/
│   ├── features/                       # Gherkin feature files
│   │   ├── authentication/
│   │   │   ├── login.feature
│   │   │   ├── registration.feature
│   │   │   └── password-reset.feature
│   │   │
│   │   ├── content/
│   │   │   ├── create-post.feature
│   │   │   ├── edit-post.feature
│   │   │   └── delete-post.feature
│   │   │
│   │   ├── api/
│   │   │   ├── users-api.feature
│   │   │   └── posts-api.feature
│   │   │
│   │   └── smoke/
│   │       └── health-check.feature
│   │
│   ├── step-definitions/               # Custom step definitions
│   │   ├── auth-steps.js               # (only if you need custom steps)
│   │   └── custom-steps.js
│   │
│   └── assets/                         # Test files (images, docs, etc.)
│       ├── test-image.png
│       ├── sample.pdf
│       └── test-data.csv
│
├── reports/                            # Generated test reports
│   └── (auto-generated)
│
└── node_modules/                       # Dependencies (auto-generated)
    └── webship-js/
        └── tests/
            └── step-definitions/
                └── webship.js          # 100+ pre-built steps
```

FIGURE 2.3 *Project Structure*

Key Principles

1. **Features organized by domain**, not by technical layer. Group by what users care about (authentication, content, shopping), not by what developers care about (controllers, models, views).
2. **Custom steps are optional.** Many projects never need custom step definitions because webship-js covers the common interactions. Only create custom steps when you have domain-specific behaviors.
3. **Assets folder for test files.** Any files your tests need to upload (images, documents, CSVs) go here.
4. **Reports are generated, not committed.** Add `reports/` to your `.gitignore`.

2.4 Configuration Deep Dive

Setup 4: The cucumber.js Configuration File

Problem: You run `npx cucumber-js` and get `0 scenarios, 0 steps`, or 47 "undefined step" warnings for steps you know exist. Both mean the same thing: Cucumber does not know where your features and step definitions live. Every webship-js project clears this hurdle exactly once, in one file, and then never thinks about it again, unless the file is wrong, in which case nothing else in this book works.

Solution:

Create `cucumber.js` in your project root:

```
module.exports = {
  default: {
    // Where to find step definitions
    requireModule: [],
    require: [
      'node_modules/webship-js/tests/step-definitions/*.js'
    ],

    // Where to find feature files
    paths: [
      'tests/features/**/*.feature'
    ],

    // webship-js configuration
    worldParameters: {
      launchUrl: 'http://localhost:8080',
      minWaitTime: {
        page: 3000,
        before_scenario: 0,
        after_scenario: 0,
        before_step: 0,
        after_step: 0
      },
      assets_folder: './tests/assets/'
    },

    // Output format
    format: ['progress-bar'],
    publishQuiet: true
  }
};
```

How It Works:

The configuration file is a CommonJS module that exports profiles. The `default` profile is used when you run `npx cucumber-js` without specifying a profile.

Let me explain each section:

requireModule: Preprocessors to load before step definitions. Empty for standard JavaScript projects. You would add `['ts-node/register']` for TypeScript.

require: Paths to step definition files. The glob `*.js` loads all JavaScript files. The order matters, webship-js steps are loaded first, then your custom steps.

paths: Where to find `.feature` files. The `**` glob pattern matches subdirectories at any depth.

worldParameters: Configuration passed to the PlaywrightWorld constructor. This is where webship-js gets its settings.

format: Output formatters. Options include:

- `progress-bar`: Shows a progress bar (good for CI)
- `progress`: Dots for each step (minimal output)
- `summary`: Just the final summary
- `json:path/to/file.json`: JSON output for reporting tools
- `html:path/to/file.html`: HTML report

publishQuiet: Suppresses the "publish your results" prompt from Cucumber.

Two defaults worth knowing because you will hit them eventually. First, the webship-js default profile sets the step timeout to **45000ms**, any single step that runs longer than 45 seconds fails the scenario, which is your safety net against a hung page stalling the whole suite. Second, the shipped configuration writes a JSON report to `tests/reports/cucumber_report.json` on every run; that file is the raw material the report generator (and your CI dashboard) reads later, so do not be surprised when it appears.

At scale: Treat `cucumber.js` as a template, not a per-project artifact. Notice that nothing in the file above is actually project-specific except `launchUrl`, and that should come from an environment variable anyway (Setup 5). When all 20 of your projects share an identical config file, "how is this suite configured?" has one answer, and a fix (say, adding a formatter) is a copy-paste rollout instead of 20 small archaeology digs.

Setup 5: Configuring the Base URL

Problem: The same suite has to run against a developer's local instance, the staging server, and, for a careful subset, production. The first instinct is to edit the config file before each run, and that instinct produces the classic accident: someone commits `launchUrl: 'https://www.example.com'`, the nightly run executes against production, and Monday starts with an incident review. Feature files must never know or care which environment they are pointed at.

Solution:

The `launchUrl` in `cucumber.js` sets the default:

```
worldParameters: {
  launchUrl: 'http://localhost:8080'
}
```

Override it per environment with the `LAUNCH_URL` environment variable:

```
# Local development
npm test

# Staging environment
LAUNCH_URL=https://staging.example.com npm test

# Production (read-only smoke tests)
LAUNCH_URL=https://www.example.com npm run test:smoke

# Using .env file pattern
export LAUNCH_URL=https://staging.example.com
npm test
```

How It Works:

webship-js resolves the base URL in this order:

1. `LAUNCH_URL` environment variable (highest priority)
2. `worldParameters.launchUrl` in `cucumber.js` (default)
3. Built-in fallback: `http://localhost:8080` if neither is set

When you write `Given I am on homepage`, webship-js navigates to this URL. When you write `When I go to "/login"`, it navigates to `{launchUrl}/login`. The resolution happens once, when the PlaywrightWorld is constructed for each scenario, so the variable must be set when the process starts, not exported mid-run.

This means your feature files are **environment-independent**:

```
# This works on localhost, staging, AND production
Given I am on homepage
When I go to "/login"
And I fill in "Email" with "test@example.com"
```

Create npm scripts for each environment:

```
{
  "scripts": {
    "test": "cucumber-js",
    "test:local": "LAUNCH_URL=http://localhost:3000 cucumber-js",
    "test:staging": "LAUNCH_URL=https://staging.example.com cucumber-js",
    "test:prod": "LAUNCH_URL=https://www.example.com cucumber-js --tags
@smoke"
  }
}
```

Notice the production script filters to `@smoke`, read-only checks only. The environment variable controls *where* tests run; tags control *which* tests are safe to run there. You need both halves of that discipline.

At scale: `LAUNCH_URL` is the single most important convention in a multi-project setup. In CI, it comes from the pipeline's environment configuration (a GitLab CI variable, a GitHub Actions environment, a Jenkins parameter), never from an edited file. That means the exact same suite, the exact same commit, produces the staging result and the production smoke result, and your delivery manager can compare them. The day a release candidate is questioned, "same tests, two URLs, both green" is a sentence everyone in the room understands.

Setup 6: Environment Variable Overrides

Problem: A test fails only in CI. To debug it locally you want Firefox instead of Chromium, a visible browser window, and slowed-down actions you can watch, but you absolutely do not want to commit any of those settings, because the next nightly run must be back to fast, headless Chromium. Config-file editing for temporary needs is how debug settings leak into main.

Solution:

Use environment variables for all dynamic settings:

```
# Base URL
export LAUNCH_URL=https://staging.example.com

# Browser choice (chromium / firefox / webkit)
export BROWSER=firefox

# Headed mode ('false' opens a visible browser window)
export HEADLESS=false

# Slow motion in ms (for debugging)
export SLOW_MO=1000

# Run tests
npm test
```

How It Works:

These variables are read by `playwright.config.ts` when the browser launches. `BROWSER` accepts `chromium`, `firefox`, or `webkit`. `HEADLESS` is headed only when set to the literal string `'false'`. `SLOW_MO` inserts a pause between every Playwright action; if you do not set it, webship-js picks a sensible default for the context, 0 in CI (full speed), 300ms when running headless locally, and 800ms in headed mode so your eyes can follow what the browser is doing. Because everything is an environment variable, your debugging session leaves no trace in the repository: close the terminal and the suite is back to its defaults.

Pro Tip: Combine them in one line for a debug run: `HEADLESS=false SLOW_MO=1000 BROWSER=firefox npx cucumber-js tests/features/auth/login.feature`. One command, one feature file, watchable speed, gone when you are done.

Setup 7: Configuring Wait Times

Problem: Your application loads in under a second on localhost and in four seconds on the shared staging server every team in the company deploys to. The same suite that is rock solid locally starts failing intermittently on staging, and the first reflex, sprinkling sleep statements through the scenarios, is exactly the move that turns a fast suite into a slow, still-flaky one.

Solution:

```
worldParameters: {
  launchUrl: 'http://localhost:8080',
  minWaitTime: {
    page: 3000,          // Wait after page navigation (ms)
    before_scenario: 0,  // Wait before each scenario
    after_scenario: 0,   // Wait after each scenario
    before_step: 0,      // Wait before each step
    after_step: 0        // Wait after each step
  }
}
```

Guidelines:

SETTING	DEFAULT	WHEN TO CHANGE
page	3000ms	Increase for slow SPAs,
		decrease for static sites
before_scenario	0ms	Increase if scenarios need
		time to clean up state
after_scenario	0ms	Increase if you need to
		capture screenshots/logs
before_step	0ms	Increase for visual debugging
		(watch each step execute)
after_step	0ms	Increase for slow applications
		that need settling time

How It Works:

Here is the part that surprises people coming from older frameworks: these are budgets, not sleeps. The `page: 3000` value is the per-navigation budget for **smartSettle**, webship-js's smart-wait engine. On every page load, an init script installs three counters in the browser: `window.__webshipAjaxCount` (wraps fetch and XHR, counts in-flight requests), `window.__webshipPendingTimers` (wraps setTimeout), and `window.__webshipLastMutation` (a MutationObserver timestamp). After navigation, smartSettle polls those counters every 50ms and declares the page "settled" the moment AJAX hits zero, timers hit zero, and the DOM has been quiet for 250ms, or gives up when the budget runs out and lets the next step proceed anyway. A static site settles in milliseconds; a chatty SPA uses the full window only when it genuinely needs it. The same engine runs automatically after every state-changing step with a 1500ms budget, which is why most scenarios need no explicit waits at all. If you ever need to prove a timing bug is not smartSettle's doing, disable it with `WEBSHIP_AUTO_SETTLE=off`.

Pro Tip: For local development, keep wait times low for fast feedback. For CI against a shared staging server, you might need to increase `page` to 5000-8000ms, and because it is a budget, the increase costs you nothing on pages that settle quickly. Resist touching `before_step` / `after_step`: those ARE static delays, multiplied by every step in the suite. A 500ms `after_step` on a 2,000-step suite is 16 minutes of pure waiting.

Setup 8: Setting the Assets Folder

Problem: The profile-picture upload test works on your machine because the test image happens to sit in your Downloads folder. The new hire clones the repo, runs the suite, and the upload scenarios fail with file-not-found. Test files are test fixtures, they belong in the repository, in one agreed place, referenced by relative path.

Solution:

```
worldParameters: {
  launchUrl: 'http://localhost:8080',
  assets_folder: './tests/assets/'
}
```

Populate the folder with test files:

```
tests/assets/
├── images/
│   ├── test-avatar.png              # 100x100 profile picture
│   ├── large-image.jpg              # Test large file handling
│   └── test-logo.svg                # SVG upload test
├── documents/
│   ├── sample.pdf                   # PDF upload test
│   ├── test-spreadsheet.xlsx        # Excel upload test
│   └── readme.txt                   # Plain text upload test
└── media/
    ├── test-video.mp4               # Video upload test
    └── test-audio.mp3               # Audio upload test
```

FIGURE 2.4 *Setup 8: Setting the Assets Folder*

Use them in features:

```
When I attach the file "images/test-avatar.png" to "profile_picture"
When I attach the file "documents/sample.pdf" to "resume"
```

Pro Tip: Keep assets small and synthetic. A 100x100 PNG tests an avatar upload as well as a 4MB photo does, and across hundreds of CI runs the difference shows up in checkout times and runner storage. The exception: keep exactly one deliberately large file for the “rejects oversized uploads” scenario.

Setup 9: Browser Configuration

Problem: A customer files a bug that only happens in Safari, and your suite has been quietly testing Chromium-only since day one. Meanwhile the platform team asks what flags the browser launches with inside their Docker runners, and nobody on the QA side can answer. Before you can debug either conversation, you need to know exactly what webship-js launches by default, which browser, which mode, which arguments, and which knobs change it.

Solution:

The Playwright configuration can be customized. webship-js uses these defaults:

```
Default Playwright Configuration in webship-js:

Browser:          Chromium (BROWSER env to change)
Headless:         true (HEADLESS='false' for headed)
Slow Motion: 0 in CI / 300ms headless / 800ms headed
Viewport:         null (uses the real window size)
HTTPS:            ignoreHTTPSErrors: true
Args:             --no-sandbox
                  --disable-setuid-sandbox
                  --disable-dev-shm-usage
                  --disable-accelerated-2d-canvas
                  --no-first-run
                  --no-zygote
                  --disable-gpu
```

FIGURE 2.5 *Setup 9: Browser Configuration*

How It Works:

Three of these defaults deserve a closer look. The slow-motion value adapts to context: 0 in CI because nobody is watching and speed is money, 300ms for local headless runs, 800ms when headed so a human can follow along, override any of them with `SLOW_MO`. The `viewport: null` setting means the page gets the browser window's actual size rather than an emulated fixed viewport; when you need specific dimensions, the responsive steps and the named breakpoints (xs through xxxl) handle that explicitly per scenario. And `ignoreHTTPSErrors: true` exists because staging environments live on self-signed certificates, without it, half your environments would fail at the TLS handshake before the first step ran.

At scale: The browser args list is tuned for containerized CI (`--no-sandbox`, `--disable-dev-shm-usage` are the classic fixes for Chromium dying inside Docker with cryptic crashes). If your platform team builds custom runner images, keep these defaults, removing them is the most com-

mon cause of “the suite segfaults only on the new runners.”

Setup 10: Headless vs Headed Mode

Problem: A scenario fails and the error message tells you what was missing but not why. You need to watch the browser do the thing, see the modal that did not close, the spinner that never stopped. But headed mode must stay a local-only habit: CI runners have no display, and a teammate who pulls your branch should never inherit a config that pops browser windows.

Solution:

By default, webship-js runs headless (no visible browser). For debugging, you can control this with environment variables or by modifying the Playwright config.

For CI:

```
# Headless (default, no visible browser)
npm test
```

For local debugging:

```
# Headed mode (opens a visible browser window)
HEADLESS=false npm test
```

Watching the browser execute your steps is incredibly useful when a test fails and you cannot figure out why. You can see exactly what the browser sees, including timing issues and element visibility problems. Headed mode also changes the pace for you: the slow-motion default rises to 800ms between actions, so the run is watchable without you setting anything.

Pro Tip: When the failure only happens in CI and you cannot reproduce it headed, do not guess, turn on video. `WEBSHIP_VIDEO=on-failure` records every scenario and keeps the videos only for the ones that fail. It is the closest thing to standing behind the CI runner watching its screen. Chapter 46 covers video recording in depth.

2.5 Running Your First Test

Setup 11: Hello World Feature

Problem: You have installed packages, downloaded browsers, and written a config file, and so far nothing has proven any of it actually works together. Before you invest a day writing real scenarios, you want one two-line test that exercises the whole chain: config, step matching, browser launch, navigation, assertion. If this passes, every layer underneath is wired correctly.

Solution:

Create `tests/features/smoke/hello-world.feature`:

```
Feature: Hello World
  As a developer
  I want to verify webship-js is working
  So that I can start writing real tests

  @smoke
  Scenario: Homepage loads
    Given I am on homepage
    Then I should see "Welcome"
```

Run it:

```
npm test
```

Expected Output (Pass):

```
1 scenario (1 passed)
2 steps (2 passed)
0m4.523s
```

Expected Output (Fail):

```
1 scenario (1 failed)
2 steps (1 passed, 1 failed)

Failing steps:
  Then I should see "Welcome"

  AssertionError: Expected to find "Welcome" on the page
    at World.Then (/node_modules/webship-js/tests/step-
definitions/webship.js:XXX)
```

If the test fails, check:

1. Is your application running at the `launchUrl`?
2. Does the homepage actually contain the text “Welcome”?
3. Is the URL correct?

How It Works:

Those two innocent lines exercise more machinery than they let on. `Given I am on homepage` makes Cucumber construct a fresh PlaywrightWorld for the scenario, launch the browser, open a new context and page, navigate to the resolved `launchUrl`, and run smartSettle until the page goes quiet (or the 3000ms navigation budget elapses). `Then I should see "Welcome"` then asserts against the visible page text. Roughly four seconds for two steps sounds slow until you realize most of it is one-time browser startup, additional scenarios in the same run reuse the launched browser and are much cheaper.

Setup 12: Understanding Test Output

Problem: The first time a teammate runs the suite, they read `12 steps (10 passed, 1 failed, 1 skipped)` and ask you whether “skipped” is something they broke. Output literacy sounds trivial until you watch someone spend twenty minutes investigating a skipped step that was never the problem, the failure one line above it was.

Solution:

Cucumber provides several output formats. Here is what each tells you:

Progress bar format (default):

```
Feature: Login ..F..

Failures:
1) Scenario: Failed login
   Step: Then I should see "Invalid credentials"
   Error: Expected "Invalid credentials" but found "Error occurred"

3 scenarios (2 passed, 1 failed)
12 steps (10 passed, 1 failed, 1 skipped)
0m12.456s
```

Understanding the symbols:

```
.  = Passed step
F  = Failed step
-  = Skipped step (after a failure)
P  = Pending step (step definition exists but not implemented)
U  = Undefined step (no step definition found)
```

Setup 13: Reading Error Messages

Problem: It is 4:50 p.m., the pipeline is red, and the error says `Expected to find 'Dashboard' on the page`. Is the application broken? Is the test wrong? Is staging down? Every minute spent decoding the message is a minute the release waits. The good news: webship-js failures cluster into a handful of patterns, and each one points at a different culprit.

Solution:

Common error messages and their meanings:

```
Error: "Expected to find 'X' on the page"
Cause: The text X does not appear on the visible page.
Fix:       - Use a web-first assertion ("should have text")
             with a "within N seconds" budget for slow content
           - Check the spelling of the expected text
           - Check that the content is visible (not hidden)

Error: "Element not found: [selector]"
Cause: Playwright cannot find the element on the page.
Fix:       - Check the field label/name matches
           - Wait for the element ("should be visible") first
           - Verify the page loaded correctly

Error: "Navigation timeout of 30000ms exceeded"
Cause: The page did not load within 30 seconds.
Fix:       - Check the URL is correct
           - Check the application is running
           - Increase timeout if the page is legitimately slow

Error: "Step 'X' is undefined"
Cause: No step definition matches the step text.
Fix:       - Check spelling matches a webship-js step exactly
           - Create a custom step definition
           - Check the require path in cucumber.js

Error: "net::ERR_CONNECTION_REFUSED"
Cause: Cannot connect to the launchUrl.
Fix:       - Start your application server
           - Check the port number matches
           - Check no firewall is blocking the connection
```

FIGURE 2.6 *Setup 13: Reading Error Messages*

How It Works:

The diagnostic order matters. Connection refused and navigation timeouts are environment problems, the application or network, not your test. Undefined steps are wiring problems, your `cucumber.js` require paths or a typo in the step text. Only the first two ("expected to find", "element not found") are genuine test-versus-application questions, and webship-js answers most of them for you: every failed step automatically captures a screenshot (`WEBSHIP_SCREENSHOT_ON_FAILED` defaults to `true`, saved with a `failed_` prefix under `./screenshots`). Open the screenshot before you open the code, nine times out of ten it shows an error page, a still-loading spinner, or a cookie-consent overlay sitting exactly where your element should be.

At scale: Teach this triage order to every project team, because it determines who gets paged. Connection errors go to DevOps, undefined steps go to whoever changed the test code, assertion failures go to QA plus the feature developer. A failure-routing convention sounds bureaucratic until the morning 6 of your 40 nightly suites are red and all six turn out to be one staging outage, which a thirty-second look at the error class would have told you.

2.6 Multiple Configuration Profiles

Setup 14: Named Profiles for Different Scenarios

Problem: The same suite serves three masters with three clocks. The developer pushing a hotfix wants a two-minute smoke signal. The nightly job wants everything except work-in-progress, with reports. The API checks should run on every commit because they need no browser and finish in seconds. One configuration cannot satisfy all three, and maintaining three config files means the day someone fixes a path in one, the other two quietly rot.

Solution:

```
// cucumber.js
const common = {
  requireModule: [],
  require: [
    'node_modules/webship-js/tests/step-definitions/*.js',
    'tests/step-definitions/*.js'
  ],
  worldParameters: {
    launchUrl: process.env.LAUNCH_URL || 'http://localhost:8080',
    assets_folder: './tests/assets/'
  },
  publishQuiet: true
};

module.exports = {
  // Default: run everything
  default: {
    ...common,
    paths: ['tests/features/**/*.feature'],
    format: ['progress-bar']
  },

  // Smoke: quick health checks
  smoke: {
    ...common,
    paths: ['tests/features/**/*.feature'],
    tags: '@smoke',
    format: ['progress-bar']
  },

  // API: only API tests (no browser needed)
  api: {
    ...common,
    paths: ['tests/features/api/**/*.feature'],
    format: ['progress-bar']
  },

  // CI: full suite with reports
  ci: {
    ...common,
    paths: ['tests/features/**/*.feature'],
```

```
    tags: 'not @wip',
    format: [
      'progress-bar',
      'json:reports/cucumber-report.json',
      'html:reports/cucumber-report.html'
    ]
  }
};
```

Run specific profiles:

```
# Default profile
npx cucumber-js

# Smoke tests
npx cucumber-js --profile smoke

# API tests only
npx cucumber-js --profile api

# CI profile with reports
npx cucumber-js --profile ci
```

Add to `package.json`:

```
{
  "scripts": {
    "test": "cucumber-js",
    "test:smoke": "cucumber-js --profile smoke",
    "test:api": "cucumber-js --profile api",
    "test:ci": "cucumber-js --profile ci"
  }
}
```

How It Works:

The `common` object plus spread syntax (`...common`) is the whole trick: shared settings live in one place, and each profile only declares what makes it different, a tag filter, a path restriction, extra formatters. Note how the `ci` profile adds the JSON formatter: that JSON file is what report generators consume, so CI is the profile where it belongs. Filtering by tags rather than file paths (the `smoke` profile scans all features but runs only `@smoke` scenarios) means a scenario's membership in the smoke suite is declared next to the scenario itself, where reviewers can see it.

At scale: Standardize the profile names across every project before you standardize anything else. When `npm run test:smoke` means the same thing in all 20 repos, your CI templates become trivial, one pipeline definition, parameterized by project, instead of 20 bespoke YAML files. It also makes the cross-project dashboard honest: "smoke" is comparable across products only if every product defines it the same way. The day a new project joins, it inherits the profiles, the scripts, and the pipeline in one copy.

Chapter Summary

- webship-js = Cucumber-js + Playwright + 411 pre-built step definitions in 36 step files
- Installation is three commands: `npm install`, `npx playwright install`, create config
- The `cucumber.js` config file controls paths, URLs, wait times, and output
- `launchUrl` is the most important setting, override with `LAUNCH_URL` env var
- Feature files go in `tests/features/`, organized by domain area
- Each scenario gets a fresh browser context (complete isolation)
- Use named profiles for different testing scenarios (smoke, full, api, ci)

In the next chapter, we will master Gherkin, the language that makes all of this possible.

Chapter 3: Gherkin Deep Dive

Gherkin is the language that makes BDD accessible to everyone. It is small, structured, and designed to be read by humans first, parsed by machines second. In this chapter, we will explore every feature of Gherkin with practical examples.

3.1 The Gherkin Language

Gherkin is not a programming language. It is a **specification language**. Its purpose is to describe software behavior in a way that is simultaneously:

- Readable by business people
- Writable by testers and product owners
- Executable by Cucumber

The Complete Keyword Reference

KEYWORD	PURPOSE
Feature	Groups related scenarios, describes the feature
Rule	Groups scenarios under a business rule
Background	Shared setup steps for all scenarios
Scenario	A single concrete example (test case)
Scenario Outline	A template with placeholders for data
Examples	Data table for Scenario Outline
Given	A precondition (setup the system state)
When	An action (the trigger being tested)
Then	An expected outcome (assertion)
And	Continues the previous Given/When/Then
But	Continues with a contrast (still same type)
*	Wildcard step (matches any keyword)
@tag	Metadata for filtering and hooks
#	Comment (ignored by Cucumber)
"""	Doc String delimiter (multiline text)

Feature File Anatomy

```
@tag1 @tag2                          # Tags (optional)
Feature: Feature Title               # Required
  As a <role>                        # Description (optional)
  I want to <action>                 #   - free-form text
  So that <business value>           #   - not parsed by Cucumber

  Background:                        # Shared setup (optional)
    Given some common precondition   #   - runs before EACH scenario

  @tag3                              # Scenario-level tags
  Scenario: Scenario Title           # A concrete example
    Given a precondition             # Setup
    When I do something              # Action
    Then something should happen     # Expected outcome

  Scenario Outline: Template         # Parameterized scenario
    Given a <thing>                  # Placeholders in angle brackets
    When I do <action>
    Then I should see <result>

    Examples:                        # Data table
      | thing  | action | result  |  # Header row
      | apple  | eat    | eaten   |  # Each row = one scenario run
      | banana | peel   | peeled  |
```

3.2 Features and Scenarios

Recipe 1 Writing a Complete Feature

Problem: The shopping cart breaks for the second sprint in a row, and the post-mortem reveals there was exactly one cart test, the happy path someone wrote on day one. Nobody ever sat down and asked "what are ALL the behaviors this cart promises?" You need a feature file that covers the promises, not just the demo, and you need its shape to be something the next person can extend without inventing their own structure.

Solution:

```
Feature: Shopping Cart
  As a customer browsing the online store
  I want to add products to my shopping cart
  So that I can purchase multiple items at once

  Scenario: Add a single item to empty cart
    Given I am on homepage
    When I go to "/products/laptop"
    And I press "Add to Cart"
    And wait for AJAX to finish
    Then I should see "1 item in cart"

  Scenario: Add multiple different items
    Given I am on homepage
    When I go to "/products/laptop"
    And I press "Add to Cart"
    And wait for AJAX to finish
    When I go to "/products/mouse"
    And I press "Add to Cart"
    And wait for AJAX to finish
    Then I should see "2 items in cart"

  Scenario: Remove item from cart
```

```
    Given I am on homepage
    When I go to "/products/laptop"
    And I press "Add to Cart"
    And wait for AJAX to finish
    When I go to "/cart"
    And I press "Remove"
    And wait for AJAX to finish
    Then I should see "Your cart is empty"

  Scenario: Cart shows total price
    Given I am on homepage
    When I go to "/products/laptop"
    And I press "Add to Cart"
    And wait for AJAX to finish
    When I go to "/cart"
    Then I should see "Total:"
    And I should see "$"
```

How It Works:

Each scenario is completely independent. Scenario 2 does not depend on scenario 1. Scenario 3 adds an item and then removes it within the same scenario. This independence is critical for reliable tests, and webship-js enforces it physically: every scenario gets a fresh browser context, so cookies, localStorage, and the cart itself start empty each time. A scenario that "worked because the previous one left an item in the cart" is impossible by construction.

The `wait for AJAX to finish` step is an explicit call to smartSettle with a 10-second budget, it watches the page's in-flight request counter (`window.__webshipAjaxCount`), pending timers, and DOM mutations, and returns the moment all three go quiet. Strictly speaking you often do not need it, because webship-js already auto-settles after every state-changing step; keeping it after a click that triggers a cart update is a readable way to say "this action is asynchronous, and we know it."

The Feature description (As a / I want / So that) is free-form text that Cucumber ignores during execution. It serves as documentation for humans, and it shows up in generated reports, where a manager scanning results reads it as the answer to "what does this feature do?"

At scale: One behavior per scenario is the rule that keeps this maintainable, it is a hard rule in the webship-js repo itself. The temptation on a deadline is one mega-scenario that adds, removes, and checks totals in 30 steps. Resist it: when the mega-scenario fails, the report says “cart is broken” and someone still has to debug which of five behaviors died. Four small scenarios fail precisely, and across 20 projects, precise failures are what make a 7 a.m. triage of overnight runs take minutes instead of hours.

Recipe 2 The User Story Format

Problem: Six months from now, someone will open this feature file during an incident and need to know in ten seconds what it protects and why it matters. If the file starts with `Feature: test_login_v2_final`, they get nothing. The three description lines under the Feature title are where you buy that future reader their ten seconds, and where the product owner’s intent survives after the PO has moved to another project.

Solution:

The standard format is:

```
Feature: <Title>
  As a <role>
  I want to <capability>
  So that <business benefit>
```

Real examples:

```
Feature: Password Reset
  As a forgetful user
  I want to reset my password via email
  So that I can regain access to my account

Feature: Order Tracking
  As a customer who placed an order
  I want to track my shipment in real time
  So that I know when to expect delivery

Feature: Admin User Management
  As a system administrator
  I want to create and manage user accounts
  So that I can control who has access to the system

Feature: Product Search
  As a shopper looking for specific products
  I want to search by keyword and filter by category
  So that I can find what I need quickly
```

Pro Tip: If you cannot articulate the “So that” (business benefit), question whether the feature needs to exist. Every feature should deliver measurable value.

At scale: The description block is also your prioritization metadata. When the CI budget forces a choice about which suites run on every commit versus nightly, “As a customer / So that I can purchase” outranks “As an admin / So that reports are sorted”, and the ranking is readable by the delivery manager making the call, not just the engineer who wrote the test. Write the business value down once, in the file, and you never have to re-litigate it in a meeting.

Recipe 3 Multiple Scenarios, The Happy Path and Beyond

Problem: Your suite is green for months, then a user pastes `<script>` into the newsletter box, or submits the form empty, and production throws a 500. The happy path was tested; nothing else was. The hard part is not writing failure scenarios, it is remembering to ask for them, systematically, for every feature, before the incident does the asking.

Solution:

Always start with the **happy path** (the expected, successful flow), then add failure scenarios and edge cases:

```
Feature: Newsletter Subscription
  As a visitor
  I want to subscribe to the newsletter
  So that I receive updates about new products

  # Happy path -- the expected flow
  Scenario: Successful subscription with valid email
    Given I am on homepage
    When I fill in "Email" with "reader@example.com"
    And I press "Subscribe"
    And wait for AJAX to finish
    Then I should see "Thank you for subscribing!"

  # Failure: invalid input
  Scenario: Subscription with invalid email
    Given I am on homepage
    When I fill in "Email" with "not-an-email"
    And I press "Subscribe"
    Then I should see "Please enter a valid email address"

  # Failure: empty input
  Scenario: Subscription with empty email
    Given I am on homepage
    When I press "Subscribe"
    Then I should see "Email is required"
```

```
# Edge case: already subscribed
Scenario: Subscription with already subscribed email
  Given I am on homepage
  When I fill in "Email" with "existing@example.com"
  And I press "Subscribe"
  And wait for AJAX to finish
  Then I should see "This email is already subscribed"

# Edge case: XSS attempt
Scenario: Subscription with script injection
  Given I am on homepage
  When I fill in "Email" with "<script>alert('xss')</script>"
  And I press "Subscribe"
  Then I should see "Please enter a valid email address"
  And I should not see "alert"
```

How It Works:

The order is deliberate: happy path first, failures second, edge cases last. The happy path documents the feature's purpose; the failure scenarios document its contract under bad input; the edge cases document the conversations your Three Amigos session surfaced. Notice which scenarios keep the `wait for AJAX to finish` step, only the ones where the server round-trip happens. Client-side validation ("not-an-email") fires instantly, so the scenario asserts immediately and the web-first text check does any remaining waiting itself.

At scale: Make the five-scenario shape (happy, invalid, empty, duplicate, injection) a house pattern for every input form across your projects. Reviewers stop checking "are there enough tests?" and start checking "which of the five is missing, and is that omission deliberate?" That question fits in a PR comment. The new hire's first feature file gets reviewed against a pattern instead of a feeling.

3.3 Background: Shared Setup

Recipe 4 Using Background for Common Steps

Problem: Your admin feature file has nine scenarios and every single one opens with the same three lines: log in, navigate, wait. Then the login flow changes, and you edit nine copies, and miss one, and spend an afternoon on a “failure” that was just a stale setup line. Repeated setup is not only noise; it is nine chances for the copies to drift apart.

Solution:

```
Feature: Blog Post Management
  As an admin
  I want to manage blog posts
  So that I can keep the website content fresh

  Background:
    Given I am a logged in user with the "admin" user
    And I go to "/admin/posts"
    And wait for AJAX to finish

  Scenario: View list of posts
    Then I should see "All Posts"
    And I should see "Create New Post"

  Scenario: Create a new blog post
    When I press "Create New Post"
    And wait for AJAX to finish
    And I fill in "Title" with "Testing with webship-js"
    And I fill in the WYSIWYG field "Body" with the "<p>This is a great
tool for BDD.</p>"
    And I press "Save"
    And wait for AJAX to finish
    Then I should see "Post created successfully"

  Scenario: Edit an existing post
    When I click "Edit" in "First Post" row
    And wait for AJAX to finish
    And I fill in "Title" with "Updated Title"
    And I press "Save"
    And wait for AJAX to finish
    Then I should see "Post updated successfully"

  Scenario: Delete a post
    Given I will accept the next dialog
    When I click "Delete" in "Old Post" row
    And wait for AJAX to finish
    Then I should not see "Old Post"
```

How It Works:

The Background runs before **every** scenario. In this example, every scenario starts logged in and on the admin posts page. Without Background, each scenario would repeat those three lines.

Three steps here deserve a closer look. `I am a logged in user with the "admin" user` is a custom step, the kind you will build yourself in Chapter 17 (the built-in route is the auth-state steps, which save a logged-in session once and restore it per scenario). The WYSIWYG step targets CKEditor and contenteditable fields, which ordinary fill steps cannot reach because the editable area is not a normal input. And the dialog step comes **before** the click on purpose: Playwright dialogs must be armed in advance. `Given I will accept the next dialog` registers a one-shot handler; when the Delete button fires `confirm()`, the handler accepts it. Put the step after the click and the dialog has already appeared with nobody listening, the scenario hangs until the 45-second step timeout kills it.

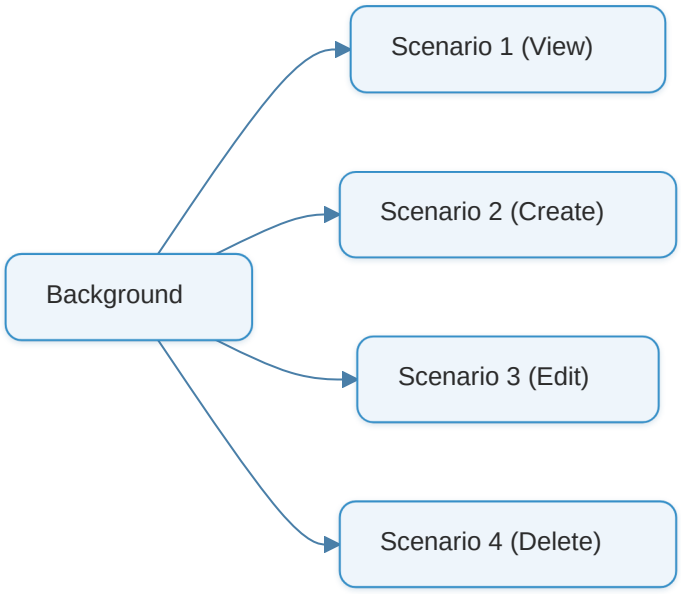


FIGURE 3.1 *Using Background for Common Steps (Recipe 4)*

At scale: Background runs are multiplication you pay for in CI minutes. A 10-second login Background in front of 15 scenarios is 2.5 minutes per feature, per project, per nightly run. This is exactly what the built-in auth-state steps exist for: log in once, `save the auth state` to a

JSON file, and let Background `restore the auth state`, a storageState restore takes milliseconds instead of a full login round-trip. Across 40 nightly suites, this one change is usually the single largest runtime saving available.

Recipe 5 When Background Gets Too Long

Problem: A teammate opens your feature file and has to scroll past eight Background lines before reaching the first scenario, and by then they have forgotten what the feature was about. Worse, every scenario now implicitly depends on all eight lines, so any scenario that does not need the full setup still pays for it on every run. A long Background is the Gherkin equivalent of a function with too many side effects.

Solution:

If Background exceeds 4-5 lines, it is a sign you should either:

Option A: Use a declarative step that hides the details:

```
# Instead of this:
Background:
  Given I am on homepage
  When I go to "/login"
  And I fill in "Email" with "admin@example.com"
  And I fill in "Password" with "AdminPass123"
  And I press "Sign In"
  And wait for AJAX to finish
  And I go to "/admin/posts"
  And wait for AJAX to finish

# Use this:
Background:
  Given I am a logged in user with the "admin" user
  And I go to "/admin/posts"
  And wait for AJAX to finish
```

Option B: Split the feature into smaller, focused files:

```
# Instead of one large feature:
tests/features/admin/posts.feature (15 scenarios)

# Split into focused features:
tests/features/admin/posts-create.feature (4 scenarios)
tests/features/admin/posts-edit.feature (4 scenarios)
tests/features/admin/posts-delete.feature (3 scenarios)
tests/features/admin/posts-list.feature (4 scenarios)
```

Each smaller feature can have a shorter, more specific Background.

How It Works:

Option A trades step count for vocabulary: one declarative step hides six imperative ones behind a name everyone understands. The detail does not disappear, it moves into a step definition where it is written once and fixed once. Option B trades file size for focus: smaller files mean smaller Backgrounds, faster targeted runs (`npx cucumber-js tests/features/admin/posts-create.feature`), and cleaner failure reports. Use both. The signal to act is the day you catch yourself scrolling to find a scenario.

3.4 Scenario Outline and Examples

Scenario Outline is one of the most powerful Gherkin features. It lets you write one template and run it with many data sets.

Recipe 6 Data-Driven Tests with Scenario Outline

Problem: The validation spec lists six input cases. The first instinct is copy-paste: six nearly identical scenarios where only the numbers differ. Then the form's button label changes and you fix six copies; then a reviewer cannot tell whether scenario four differs from scenario five on purpose or by accident. Duplicated scenarios are where feature files go to rot.

Solution:

```
Feature: Calculator
  As a user
  I want to perform calculations
  So that I get accurate results

  Scenario Outline: Addition of two numbers
    Given I am on homepage
    When I go to "/calculator"
    And I fill in "First Number" with "<num1>"
    And I fill in "Second Number" with "<num2>"
    And I press "Add"
    Then I should see "<result>"

    Examples:
      | num1 | num2 | result |
      | 1    | 1    | 2      |
      | 5    | 3    | 8      |
      | 10   | 20   | 30     |
      | 0    | 0    | 0      |
      | -5   | 5    | 0      |
      | 100  | 200  | 300    |
```

How It Works:

Cucumber generates one scenario per row in the Examples table. The `<num1>`, `<num2>`, and `<result>` placeholders are replaced with the actual values. This example generates **6 scenarios** from a single template.

Each generated scenario is a full, independent run, fresh browser context, fresh navigation, fresh smartSettle cycle. That honesty has a price: 6 rows cost 6 browser sessions. Keep outline rows for cases that genuinely differ in behavior (zero, negatives, boundaries), not for bulk, if you want to hammer an algorithm with 500 inputs, that is a unit test's job, not an acceptance suite's.

Without Scenario Outline, you would need to write 6 separate scenarios, each identical except for the numbers. That is exactly the kind of duplication that BDD was designed to eliminate.

Pro Tip: When a row fails, Cucumber reports it with the row's values inlined, so the failure reads "Addition of two numbers, Examples row `| -5 | 5 | 0 |`", you know instantly which input broke without re-running anything.

Recipe 7 Multiple Examples Tables

Problem: Your validation outline has a dozen rows, and when row eight fails at 2 a.m., the on-call engineer stares at `| user+tag@example.com | Registration successful |` with no idea whether that row tests a valid input or an invalid one. A flat wall of data rows hides the intent that made each row worth testing.

Solution:

```
Feature: User Registration Validation
  As a new user
  I want clear error messages during registration
  So that I can correct my information

  Scenario Outline: Email validation
    Given I am on homepage
    When I go to "/register"
    And I fill in "Email" with "<email>"
    And I press "Register"
    Then I should see "<message>"

    Examples: Invalid emails
      | email              | message                    |
      | not-an-email       | Please enter a valid email |
      | missing@           | Please enter a valid email |
      | @no-local-part.com | Please enter a valid email |
      | spaces in@email.com| Please enter a valid email |

    Examples: Valid emails
      | email                  | message                    |
      | user@example.com       | Registration successful    |
      | user+tag@example.com   | Registration successful    |
      | user@subdomain.example.com | Registration successful |
```

How It Works:

You can have multiple Examples sections with different labels. The labels ("Invalid emails", "Valid emails") are just documentation, they appear in test output and in generated HTML reports, so a failure identifies its group without anyone opening the file. The grouping also keeps reviews honest: when a developer adds a new accepted email format, the PR diff shows a row landing in "Valid emails," which is exactly the kind of behavioral change a product owner should see and approve.

Pro Tip: You can also tag individual Examples blocks (`@valid` above one table, `@invalid` above another) and run just one group with `--tags`. Useful when the invalid-input rows are fast client-side checks and the valid rows hit the real registration backend you only want to exercise

nightly.

Recipe 8 Complex Data Combinations

Problem: Three roles, four protected areas, and the access rules live in a wiki table nobody has compared to the actual application since the security review last year. Then an editor reports they can suddenly see the user-management page. Permission matrices are the classic silent-regression zone: one changed default in an upgrade flips a cell, and no single feature “looks” broken.

Solution:

```
Feature: Role-Based Access Control
  As an admin
  I want different pages to be accessible by different roles
  So that sensitive data is protected

  Scenario Outline: Access control per role
    Given I am a logged in user with the "<role>" user
    When I go to "<page>"
    Then I should see "<expected>"

    Examples: Admin can access everything
      | role  | page            | expected           |
      | admin | /admin/users    | User Management    |
      | admin | /admin/settings | System Settings    |
      | admin | /admin/reports  | Reports Dashboard  |
      | admin | /dashboard      | Dashboard          |

    Examples: Editor has limited access
      | role   | page            | expected           |
      | editor | /admin/users    | Access Denied      |
      | editor | /admin/settings | Access Denied      |
      | editor | /dashboard      | Dashboard          |

    Examples: Viewer has minimal access
      | role   | page            | expected           |
      | viewer | /admin/users    | Access Denied      |
      | viewer | /admin/settings | Access Denied      |
      | viewer | /admin/reports  | Access Denied      |
      | viewer | /dashboard      | Dashboard          |
```

This single Scenario Outline generates **10 test cases** covering the entire access control matrix. Without Scenario Outline, that would be 10 separate scenarios with nearly identical text.

3.5 Data Tables

Data Tables allow you to pass structured data to steps.

Recipe 9 Passing Tabular Data to Steps

Problem: The admin bulk-import feature takes a CSV of users, and the first attempt at testing it is a scenario with twelve fill steps and three submits, unreadable to the product owner who defined the import rules, and unreviewable in a PR diff. What you actually want is the test data laid out the way the PO already thinks about it: as a table, three users, four columns, one step. The day the import silently drops the Role column, you want the failing scenario to show the table that proves it.

Solution:

```
Feature: Bulk User Import
  Scenario: Import users from a table
    Given I am a logged in user with the "admin" user
    When I go to "/admin/users/import"
    And I fill in the form with the following data:
      | First Name | Last Name | Email              | Role   |
      | Rajab      | Natshah   | rajab@example.com  | admin  |
      | Ahmad      | Ali       | ahmad@example.com  | editor |
      | Sara       | Hassan    | sara@example.com   | viewer |
    And I press "Import Users"
    And wait for AJAX to finish
    Then I should see "3 users imported successfully"
```

How It Works:

The table rows after the step are not parsed by Cucumber into anything magical, they arrive in the step definition as a DataTable object, and calling `.hashes()` on it gives you an array of plain objects keyed by the header row (`{ 'First Name': 'Rajab', 'Last Name': 'Natshah', ... }`). That is the whole mechanism. Which means multi-column tables like this one need a custom step definition to consume them: `I fill in the form with the following data:` is exactly that kind of step, and Chapter 17 shows you how to write it in a few lines. For the simpler two-column case, field name and value, you do not need custom code at all: the built-in `When I fill in the following:` step takes a key-value table and runs each row through the same label-first field resolution as a normal fill step.

The payoff is in the failure report. When the import drops a column, the failed scenario prints the table exactly as written, the product owner reads the same three rows they specified and sees which user came out wrong. No log spelunking required.

At scale: Write the table-consuming step once and share it across projects, because bulk-import features reappear everywhere, users, products, redirects, taxonomy terms. The table format becomes a portable contract: the PO on any project fills in rows, the step already knows what to do with them. The alternative, each project inventing its own import-test phrasing, is how you end up with five steps that do the same thing and none that anyone remembers.

Recipe 10 Key-Value Data Tables

Problem: The profile form has seven fields, and the scenario that fills them is a wall of near-identical lines, the reviewer's eye slides right past the one that matters. When the PR changes "which fields does this test set?", nobody can tell at a glance. You want the form data to read like a form: field on the left, value on the right, one row each, so the diff shows exactly which field changed and the product owner can review test data without reading step syntax.

Solution:

```
Feature: User Profile
  Scenario: Update profile information
    Given I am a logged in user with the "admin" user
    When I go to "/profile/edit"
    And I fill in "First Name" with "Rajab"
    And I fill in "Last Name" with "Natshah"
    And I fill in "Email" with "rajab@example.com"
    And I fill in "Phone" with "+962-79-1234567"
    And I select "Canada" from "Country"
    And I fill in "Bio" with "Software engineer and BDD enthusiast"
    And I press "Save Profile"
    And wait for AJAX to finish
    Then I should see "Profile updated"
```

That version works, and each `And I fill in` step is explicit. But webship-js 2.0 also ships a built-in table form that says the same thing in one step:

```
  Scenario: Update profile information with a table
    Given I am a logged in user with the "admin" user
    When I go to "/profile/edit"
    And I fill in the following:
      | First Name | Rajab                                |
      | Last Name  | Natshah                              |
      | Email      | rajab@example.com                    |
      | Phone      | +962-79-1234567                      |
      | Bio        | Software engineer and BDD enthusiast |
    And I select "Canada" from "Country"
    And I press "Save Profile"
    And wait for AJAX to finish
    Then I should see "Profile updated"
```

How It Works:

`When I fill in the following:` walks the table row by row and runs each pair through the exact same field resolution as a single fill step: it looks for a label matching the left column first, then falls back to id, name, and placeholder. So the table is not a different mechanism, it is the same mechanism, batched. That matters for debugging: if “Phone” cannot be found, the failure points at that row, and the fix is the same fix you would make for a standalone fill step.

Two practical notes. The select stays outside the table because dropdowns go through `I select ... from ...`, not the fill logic. And the table form earns its keep at around four fields or more, below that, individual fill steps read just as well and produce more precise step-level timing in reports.

Recipe 11 Multi-Column Data Tables for Verification

Problem: The user list shows the right names and the right roles, the test passes, and the page is still wrong, because “Ahmad” is on the row that says “admin” and “Rajab” is the one marked “editor”. Plain text assertions check that words exist somewhere on the page; they say nothing about which row a word sits in. The day this bites is the day someone clicks “Delete” next to the wrong account because the table rendered shifted by one row, and your green suite swore everything was fine.

Solution:

```
Feature: User List
  Scenario: Verify user table contents
    Given I am a logged in user with the "admin" user
    When I go to "/admin/users"
    And wait for AJAX to finish
    Then I should see "Rajab"
    And I should see "admin"
    And I should see "Ahmad"
    And I should see "editor"
```

For row-specific verification:

```
Scenario: Verify specific row content
  Given I am a logged in user with the "admin" user
  When I go to "/admin/users"
  And wait for AJAX to finish
  Then I should see "Edit" in "Rajab" row
  And I should see "Delete" in "Rajab" row
```

How It Works:

The two solutions test different claims. The first scenario's `I should see` steps assert presence: each piece of text exists somewhere in the visible page. Four passing assertions there still cannot tell you that "Rajab" and "admin" share a row. The row-scoped step closes that gap: `I should see "Edit" in "Rajab" row` first locates the table row whose text contains "Rajab", then runs the assertion inside that row only. You are no longer testing that two strings exist, you are testing that they are associated, which is what a table actually promises.

Use presence checks for "did the data load at all" and row-scoped checks for "is the data attached to the right record". Most table bugs that reach production are the second kind, and they are invisible to the first kind of assertion. Chapter 9 goes deep on table testing, including cell-level and column-level checks.

At scale: Admin tables are where multi-project portfolios quietly rot, every CMS-based product has a user list, a content list, an orders list, and they all change shape during upgrades. Standardize on row-scoped assertions for anything destructive (Delete, Unpublish, Revoke) as a house rule. When the platform team rolls a framework upgrade across 20 sites, the suites that assert row association catch the shifted-column regressions; the suites that assert bare text wave them through.

Doc Strings let you pass multiline text to a step.

Recipe 12 Multiline Text with Doc Strings

Problem: The support-ticket form needs a realistic description, multiple paragraphs, a pasted error message, blank lines between them, and the only test anyone wrote fills it with “test test test”. Then a release breaks line-break handling, every ticket arrives as one run-on paragraph, and the support team finds out before the test suite does. You cannot cram a realistic block of text into a one-line quoted string; the escaping alone makes the scenario unreadable, so people simply do not test it.

Solution:

```
Feature: Support Ticket
  Scenario: Submit a detailed support ticket
    Given I am on homepage
    When I go to "/support/new-ticket"
    And I fill in "Subject" with "Installation issue"
    And I fill in "Description" with:
      """
      I am trying to install webship-js on Ubuntu 22.04.

      I ran 'npm install webship-js' and got the following error:

      Error: EACCES: permission denied, mkdir
'/usr/local/lib/node_modules'

      I have Node.js v20.11.0 installed.

      Please help.
      """
    And I press "Submit Ticket"
    And wait for AJAX to finish
    Then I should see "Ticket submitted"
```

How It Works:

Everything between the `"""` delimiters is passed to the step definition as one string argument, the last argument, after any quoted parameters in the step text. Line breaks survive exactly as written, including the blank lines between paragraphs, which is the whole point: the scenario above genuinely tests that the application preserves formatting, because the input genuinely has formatting. One detail that trips people up: Gherkin strips the indentation, using the column of the opening `"""` as the left margin. Indent the text content relative to the delimiter and it arrives flush-left in your step; indent it further and the extra spaces are kept. That lets the feature file stay visually nested without polluting the data.

Doc Strings are the right tool whenever the content IS the test:

- Long form field content (does the field preserve what users actually type?)
- JSON payloads (Recipe 13)
- HTML content for rich text editors
- Code blocks pasted into technical forms
- Exact error message text you expect the app to display

Pro Tip: Make the Doc String content realistic, not lorem ipsum. The ticket above includes a real-looking npm error because that is what users paste into support forms, and special characters in pasted errors (quotes, slashes, angle brackets) are exactly what breaks naive form handling.

Recipe 13 JSON Payloads in Doc Strings

Problem: The mobile app broke on Thursday because the backend deploy changed a response field, and nobody noticed, the browser suite was green, because the browser never talks to that endpoint the way the mobile app does. The API contract needs its own tests, and an API test needs to send a real JSON body: nested objects, arrays, numbers that must stay numbers. Building that payload out of individual form-style steps is impossible. This is where Doc Strings and the API steps meet.

Solution:

```
Feature: API - Create Product
  Scenario: Create a product via API
    Given the API base URL is "https://api.example.com"
    And I set header "Content-Type" with value "application/json"
    And I set header "Authorization" with value "Bearer test-token-123"
    When I send a POST request to "/products" with body:
      """
      {
        "name": "webship-js Pro License",
        "price": 99.99,
        "currency": "USD",
        "description": "Professional license for webship-js testing
framework",
        "category": "software",
        "tags": ["testing", "bdd", "automation"],
        "metadata": {
          "version": "2.0",
          "platform": "nodejs"
        }
      }
      """
    Then the API response code should be 201
    And the API response should contain "webship-js Pro License"
```

How It Works:

The API steps run on axios, not on the browser, no page, no Playwright, no rendering. `Given the API base URL is` sets the root every request in the scenario resolves against, and each `I set header` accumulates a header that rides along on every subsequent request in that scenario, which is exactly what you want for auth tokens. The Doc String body is sent verbatim: `99.99` stays a number and the nested `metadata` object stays nested, because nothing re-parses or re-serializes your payload on the way out. `Then the API response code should be 201` asserts the status, and `the API response should contain` does a substring check against the response body. When you need to assert structure rather than presence, "the `name` field specifically, not just the string anywhere", the JSON-pointer step (`Then the JSON response should have "/name" equal to ...`) does that, and Chapter 11 covers it along with `{{key}}` placeholders for chaining values from one response into the next request.

While you are debugging a contract, `Then print API response` dumps the full response body to the test output, crude and perfect.

At scale: API scenarios are the cheapest tests you own: no browser launch, no page settle, just an HTTP round-trip. Across a portfolio, that changes where they live in the pipeline, browser suites run nightly because they cost minutes, but API contract suites can run on every commit of every project because they cost seconds. When a backend team shares one API across four frontend products, put the contract tests in their own suite that all four pipelines trigger: one failing contract test then blocks the deploy that would have broken four products.

3.7 Tags: Organizing and Filtering

Tags are one of the most practical features of Gherkin. They let you categorize, filter, and control test execution.

Recipe 14 Tag Strategies for Large Projects

Problem: A hotfix is ready at 4 p.m. and the question is “can we get a quick confidence check before it ships?”, and the honest answer is no, because the suite is 200 scenarios with no way to ask for “just the fast, critical ones”. Or the opposite failure: every developer invented their own tags, so `@smoke` on this project means five scenarios and on the next project it means ninety. Tags are cheap to add and expensive to retrofit; the time to pick a taxonomy is before the suite grows, not after.

Solution:

Use a consistent tagging strategy:

```
# PURPOSE TAGS: Why does this test exist?
@smoke         # Quick health check
@regression    # Full regression coverage
@acceptance    # User acceptance criteria
@exploratory   # Edge cases and unusual flows

# SPEED TAGS: How long does it take?
@fast          # Under 5 seconds
@slow          # Over 30 seconds
@nightly       # Run only in nightly builds

# TECHNOLOGY TAGS: What technology is needed?
@javascript    # Requires browser automation
@api           # API-only test (no browser)
@no-javascript # Can run without a browser

# ENVIRONMENT TAGS: Where can it run?
@local         # Only works in local dev
@staging       # Only works on staging
@production    # Safe to run on production
@any           # Works anywhere

# STATUS TAGS: What is the test's state?
@wip           # Work in progress (skip in CI)
@flaky         # Known flaky (investigate)
@skip          # Temporarily disabled

# DOMAIN TAGS: What business area?
@auth          # Authentication
@checkout      # Checkout flow
@admin         # Admin functionality
@content       # Content management
@search        # Search functionality
```

Apply them:

```
@javascript @auth
Feature: User Login

  @smoke @fast
  Scenario: Login with valid credentials
    ...

  @regression @slow
  Scenario: Login with locked account
    ...

  @wip
  Scenario: Login with two-factor authentication
    ...
```

How It Works:

Tags inherit downward: a tag on the Feature line applies to every scenario in the file, and a scenario's effective tag set is the union of feature-level and scenario-level tags. In the example, the "Login with locked account" scenario carries `@javascript @auth @regression @slow`, all four are filterable.

One thing to know before you invent names: some tags already mean something to webship-js. `@javascript` (and `@js-fail`) switches JavaScript error capture into fail mode, a console error fails the scenario instead of warning. `@screenshots` turns on a screenshot after every step of that scenario. `@video` forces video recording on and `@no-video` suppresses it, regardless of the `WEBSHIP_VIDEO` setting. These doubled meanings are usually a feature, not a bug, tagging a feature `@javascript` because it is JS-heavy AND having console errors fail it is a sensible pairing, but you should choose them deliberately, not collide with them by accident.

At scale: The tag taxonomy is the single highest-return standard you can set across a portfolio, because tags are the API between your suites and everything that runs them. When `@smoke` is defined the same way on all 20 projects, say, "read-only, under 5 seconds, safe on production", then one CI template runs `--tags @smoke` everywhere, one dashboard can compare smoke health

across products, and “run the smoke suite on project X” is an instruction anyone can execute. Write the taxonomy down in a one-page document, link it from every repo, and review new tags in PRs the way you review new dependencies.

Recipe 15 Boolean Tag Expressions

Problem: Friday afternoon, a payment hotfix is ready, and what you actually want to run is a sentence: everything tagged smoke or regression, but skipping the two scenarios marked flaky and anything still work-in-progress. With single-tag runs you would fire four commands and merge the results in your head, or worse, comment scenarios out and forget to put them back. Tag expressions let you say the whole sentence in one filter.

Solution:

Cucumber supports AND, OR, and NOT expressions:

```
# Run tests tagged @smoke
npx cucumber-js --tags "@smoke"

# Run tests tagged @smoke AND @auth
npx cucumber-js --tags "@smoke and @auth"

# Run tests tagged @smoke OR @fast
npx cucumber-js --tags "@smoke or @fast"

# Run tests NOT tagged @wip
npx cucumber-js --tags "not @wip"

# Run tests tagged @smoke but NOT @slow
npx cucumber-js --tags "@smoke and not @slow"

# Complex: (smoke OR regression) AND NOT wip AND NOT flaky
npx cucumber-js --tags "(@smoke or @regression) and not @wip and not
@flaky"
```

How It Works:

The expression is evaluated per scenario against its full effective tag set, feature-level tags, scenario-level tags, and (for Scenario Outlines) tags on the specific Examples block, all merged. `and`, `or`, `not`, and parentheses compose exactly like boolean logic, and the keywords are lowercase. Always quote the whole expression: the parentheses and spaces mean something to your shell too, and an unquoted expression that "mostly works" is a classic source of CI runs that silently execute the wrong subset.

Filtering happens before execution, not during, excluded scenarios are never started, no browser launches for them, and they appear in no report. That makes `not @wip` fundamentally different from skipping: a `@wip` scenario excluded from CI costs nothing and fails nothing, which is exactly how half-written scenarios should behave until they are ready.

At scale: Raw tag expressions typed by hand are where consistency dies, one engineer types `not @wip`, another forgets, and the two runs are not comparable. Freeze the canonical expressions into npm scripts (`test:ci` = `"not @wip and not @flaky"`, `test:hotfix` = `"(@smoke or @regression) and not @wip and not @flaky"`) and make the scripts, not the expressions, the interface everyone uses. Then roll the same script names across every project: the pipeline templates call `npm run test:ci` and never need to know what the expression underneath says.

Recipe 16 Environment Tags

Problem: Someone pointed the full suite at production "just to be sure before the demo", and the payment scenarios dutifully hit the real gateway. Finance noticed before engineering did. The root cause was not carelessness, it was that nothing in the suite distinguished "safe anywhere" from "test gateway only". `LAUNCH_URL` controls where the suite runs; you also need tags that declare which scenarios are safe to run there, because a scenario that writes data is a different animal on production than on staging.

Solution:

```
@javascript
Feature: Payment Processing

  @staging @production
  Scenario: Process credit card payment
    # This uses real payment gateway - only run on staging/prod
    Given I am a logged in user with the "customer" user
    When I go to "/checkout"
    And I fill in "Card Number" with "4242424242424242"
    And I press "Pay Now"
    Then I should see "Payment successful"

  @local @staging
  Scenario: Process test payment
    # This uses test payment gateway - safe for local/staging
    Given I am a logged in user with the "customer" user
    When I go to "/checkout"
    And I fill in "Card Number" with "4000000000000002"
    And I press "Pay Now"
    Then I should see "Card declined"
```

Run per environment:

```
# Local development
npx cucumber-js --tags "@local"

# Staging
npx cucumber-js --tags "@staging"

# Production (only safe tests)
npx cucumber-js --tags "@production"
```

How It Works:

The tags carry no enforcement by themselves, `@production` does not stop anyone from running a scenario against production; it only makes the intent filterable. The mechanism is the pairing: every run combines a `LAUNCH_URL` (where) with a `--tags` filter (what is safe there), and the scenario's tag set declares which pairings its author considered. Notice the two scenarios above differ in exactly two things: the card number and the tag line. The 4242 card goes to the real gateway, so it is tagged for environments where that gateway is wired up; the decline-test card is harmless, so it runs everywhere except production. The tag line is the risk assessment, written down where the next person can see it.

Because the tags are advisory, the discipline lives in how runs are launched. Pair URL and tags in npm scripts, `"test:prod": "LAUNCH_URL=https://www.example.com cucumber-js --tags @production"`, so the safe combination is one command and the dangerous combination requires deliberately typing something nobody has aliased.

At scale: Across many projects this pairing belongs in the pipeline template, not in human hands at all. Each CI environment (staging deploy job, production post-deploy check) hard-codes its own URL-plus-tags pair, and no job exists that runs unfiltered tests against production, the dangerous combination is not just discouraged, it is unexpressible in your CI. That is also your audit answer: when someone asks "what do we run against production?", the reply is a tag expression in a versioned template, not a recollection.

3.8 Comments and Descriptions

Recipe 17 Documenting Features Effectively

Problem: Two years after this feature file was written, an auditor asks why faceted filtering is not covered, and the engineer who knew the answer left last spring. Or it is 2 a.m. during an incident and the on-call engineer opens `search.feature` needing the related ticket number and the scope

decisions, fast. Feature files outlive their authors. The context that made the scenarios make sense has to live in the file itself, or it dies with the team that wrote it.

Solution:

```
# This feature tests the search functionality.
# Related ticket: PROJ-1234
# Last reviewed: 2026-03-15

Feature: Product Search
  As a customer browsing the online store
  I want to search for products by keyword
  So that I can quickly find what I need

  The search should support:
  - Keyword matching in product titles
  - Partial word matching (e.g., "lap" matches "laptop")
  - Special character handling (no SQL injection)

  Out of scope for this feature:
  - Full-text search in descriptions (PROJ-1567)
  - Faceted filtering (PROJ-1568)

  # This description block is free text.
  # Cucumber ignores everything between the Feature line
  # and the first Scenario/Background/Scenario Outline.

  Scenario: Search by exact product name
    Given I am on homepage
    When I fill in "Search" with "MacBook Pro"
    And I press "Search"
    And wait for AJAX to finish
    Then I should see "MacBook Pro"
```

How It Works:

The two annotation channels behave differently, and the difference decides where each piece of context belongs. Comments (lines starting with `#`) are invisible everywhere except the source file, Cucumber skips them entirely, and they never reach the JSON report. Use them for maintainer-only notes: ticket references, review dates, “do not reorder these scenarios because…”. The free-text description between the Feature line and the first scenario is also ignored at runtime, but it IS captured into the JSON report and rendered by the HTML report generator, which means everything in that block is read by people who never open the repository. That is where the scope statement belongs: what the search supports, what is explicitly out of scope, and which tickets own the deferred parts. The auditor’s question above is answered by the report itself: “out of scope: faceted filtering (PROJ-1568)”, a decision, recorded, with a pointer.

One habit worth building: when a Three Amigos session decides NOT to test something, write that down in the description. Untested-by-decision and untested-by-oversight look identical in a coverage gap analysis unless the decision is recorded. Across a portfolio, a consistent header shape, ticket link, last-reviewed date, in-scope list, out-of-scope list, turns your feature files into the audit trail nobody has to reconstruct later.

3.9 Internationalization

Recipe 18 Writing Gherkin in Other Languages

Problem: Your product owner writes flawless acceptance criteria, in Arabic. Forcing the criteria through English-only Gherkin means someone translates them, and every translation is a chance for the criteria and the test to quietly diverge; the PO also stops reading the feature files, which defeats the reason you adopted BDD. I have sat in exactly this meeting. The good news: Gherkin’s keywords speak more languages than your team does, and you can keep the PO reading.

Solution:

Gherkin supports 70+ languages. Add a language header to your feature file:

Arabic:

```
# language: ar
خاصية: تسجيل الدخول
  بصفتي مستخدم مسجل
  أريد تسجيل الدخول إلى حسابي
  حتى أتمكن من الوصول إلى لوحة التحكم

  سيناريو: تسجيل دخول ناجح
    بفرض I am on homepage
    عندما I go to "/login"
    و I fill in "Email" with "user@example.com"
    و I press "Sign In"
    اذاً I should see "Dashboard"
```

French:

```
# language: fr
Fonctionnalité: Connexion
  En tant que utilisateur enregistré
  Je veux me connecter à mon compte
  Afin que je puisse accéder à mon tableau de bord

  Scénario: Connexion réussie
    Soit I am on homepage
    Quand I go to "/login"
    Et I fill in "Email" with "user@example.com"
    Et I press "Sign In"
    Alors I should see "Dashboard"
```

How It Works:

The `# language: XX` header must be the first line of the file, it switches the parser's keyword dictionary before parsing begins, so `سيناريو` and `Scénario` are recognized exactly like `Scenario`. Crucially, the keyword is the only thing being translated. Step matching ignores the keyword entirely and matches on the text after it, which is why the examples above mix Arabic keywords with English step text: `عندما I go to "/login"` resolves to the same built-in step definition as `When I go to "/login"`. That mixed mode is not a hack; it is the pragmatic sweet spot. The 411 built-in webship-js steps match English patterns, so keeping step text in English means the whole built-in library keeps working while the structure, Feature title, scenario names, descriptions, speaks the team's language.

If you want fully localized step text too, that is possible: step definitions match regular expressions, and nothing stops you writing custom steps with Arabic patterns that delegate to the same helpers. The cost is that you now maintain a translation layer of step definitions, so reserve it for teams where stakeholders truly cannot work with mixed text. Chapter 32 covers testing international and RTL sites in depth, a different problem (the SITE is multilingual) from this one (the TEAM is).

The decision that matters across a portfolio: pick one convention and apply it everywhere. Mixed-mode files (local keywords, English steps) stay portable, any QA engineer in the organization can read the executable lines of any project's suite, and the suites all share one step library. Twenty projects where three are fully localized and seventeen are not is the worst of both worlds.

3.10 Imperative vs Declarative Style

This is arguably the most important section in this entire chapter. Getting this right determines whether your test suite is maintainable or a burden.

The Spectrum

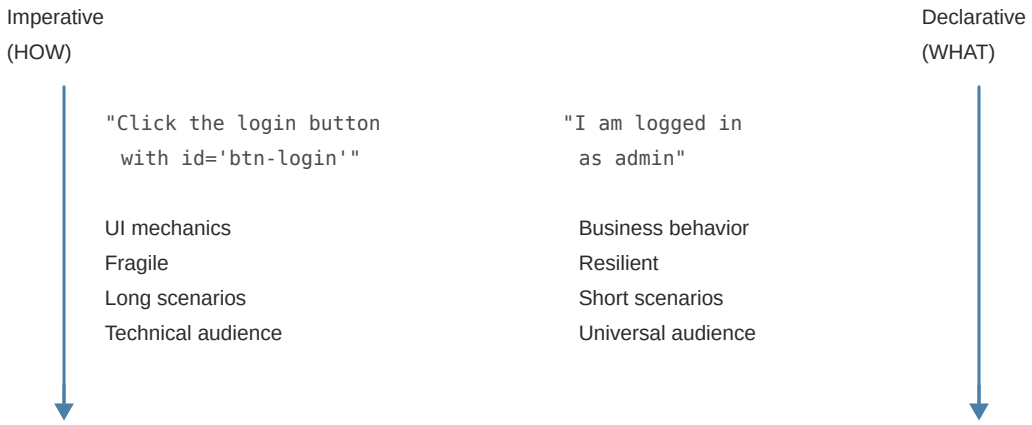


FIGURE 3.2 *The Spectrum*

Recipe 19 Converting Imperative to Declarative

Problem: The design team ships a UI refresh, same behavior, new markup, and forty scenarios go red, none of them because anything users care about broke. The post-mortem finds the scenarios were written by copying element ids out of devtools: `email-input`, `login-button`, `save-user-btn`. Every scenario that knows HOW the page is built breaks when the page is rebuilt. The new hire who wrote them did nothing unusual; imperative style is what scenarios drift into when nobody shows the team the alternative.

Solution:

Before (Imperative, 12 steps, fragile):

```
Scenario: Admin creates a new user
  Given I am on homepage
  When I go to "/login"
  And I fill in "email-input" with "admin@example.com"
  And I fill in "password-input" with "AdminPass123!"
  And I press "login-button"
  And wait for AJAX to finish
  And I go to "/admin/users"
  And wait for AJAX to finish
  And I press "create-new-user-btn"
  And I fill in "new-user-email" with "newguy@example.com"
  And I fill in "new-user-role" with "editor"
  And I press "save-user-btn"
  And wait for AJAX to finish
  Then I should see "User created"
```

After (Declarative, 5 steps, resilient):

```
Scenario: Admin creates a new user
  Given I am a logged in user with the "admin" user
  When I go to "/admin/users"
  And I press "Create New User"
  And I fill in "Email" with "newguy@example.com"
  And I select "Editor" from "Role"
  And I press "Save"
  And wait for AJAX to finish
  Then I should see "User created"
```

What Changed:

1. Login details moved to a single declarative step (`I am a logged in user`)
2. Technical IDs (`email-input`, `login-button`) replaced with human-readable labels
3. Intermediate waits removed where webship-js handles them automatically
4. The scenario now reads like a story, not a test script

How It Works:

Two pieces of webship-js machinery make the declarative version practical. First, field resolution is label-first: `I fill in "Email" with ...` looks for a form label reading “Email” before falling back to id, name, and placeholder. The label is what the user sees, and it is the most stable thing on the page, redesigns change ids and class names constantly, but renaming “Email” to something else is a product decision someone would notice in review. Writing scenarios against labels means writing them against the same contract the user relies on. Second, the auto-settle hook removes the wait bookkeeping: after every state-changing step, webship-js runs smartSettle with a 1500ms budget, waiting for in-flight requests and pending timers to drain. The imperative version’s scattered `wait for AJAX to finish` lines were mostly doing that job by hand; the declarative version keeps only the one after the final submit, where a slow server round-trip deserves an explicit, readable marker.

This is the repo rule “test behaviour, not implementation” applied at the Gherkin layer. The before-version tests the implementation (these ids, this page flow); the after-version tests the behavior (an admin can create a user). Only the second survives contact with a redesign.

At scale: Multiply the UI refresh by a portfolio. When a shared design system rolls out across 20 sites, imperative suites generate weeks of test repair, I have watched that bill arrive at Vardot when a distribution-wide theme update landed. Declarative suites built on labels and shared steps mostly do not notice: labels rarely change, and when one does, the fix is one line in one step or one selector entry, not forty scenarios. The conversion in this recipe is not a style preference; it is the difference between a portfolio that absorbs redesigns and one that bills for them.

Recipe 20 When Imperative Is Appropriate

Problem: After Recipe 19, someone on the team will swing too far and declare all detail forbidden, and then the UX regression slips through: the inline validation message stopped appearing as you type, users submit the form blind, and the declarative test (“a user can register”) still passes because registration ultimately works. When the interaction itself is the requirement, this message, at this moment, in response to this input, hiding the detail behind a declarative step would delete the very thing you are testing.

Solution:

Sometimes you WANT imperative steps, when the UI behavior IS the thing being tested:

```
Feature: Form Validation UX
  As a UX designer
  I want validation messages to appear correctly
  So that users know how to fix their input

  Scenario: Email validation shows inline error
    Given I am on homepage
    When I go to "/register"
    And I fill in "Email" with "not-valid"
    And I press "Register"
    Then I should see "Please enter a valid email"

  Scenario: Password strength indicator
    Given I am on homepage
    When I go to "/register"
    And I fill in "Password" with "weak"
    Then I should see "Weak"
    When I fill in "Password" with "StrongerPass123!"
    Then I should see "Strong"
```

How It Works:

Here the imperative details ARE the specification. “weak” and “StrongerPass123!” are not arbitrary test data, they are the boundary inputs the UX team chose, and the exact words “Weak” and “Strong” are the feedback contract. Replace any of that with a declarative abstraction and the scenario stops testing what it exists to test.

Notice the password-strength scenario asserts twice in one scenario, between actions. That works because of what happens between steps: strength indicators are typically wired to a debounced input handler, and after each fill step webship-js’s auto-settle hook waits for pending timers (`window.__webshipPendingTimers`) to drain and the DOM to go quiet before the next step runs. The assertion fires after the debounce has done its work, not in the race window before it. This is the kind of scenario that is miserably flaky in frameworks that need hand-tuned sleeps, and boringly stable here.

The Rule: Use declarative steps for setup and verification of business outcomes. Use imperative steps when the UI interaction itself is what you are testing. In practice most feature files are declarative with a small, deliberate imperative section, and it helps reviewers when the imperative ones live in a clearly named file like `form-validation-ux.feature`, so nobody “fixes” their detail away.

3.11 Advanced Gherkin Patterns

Pattern: The Narrative Scenario

Write scenarios that read like a story:

```
Feature: First-Time User Experience
  As a new user
  I want a smooth onboarding experience
  So that I understand how to use the application

  Scenario: New user signs up and completes onboarding
    Given I am on homepage
    When I click "Get Started"
    Then I should see "Create Your Account"

    When I fill in "Email" with "newuser@example.com"
    And I fill in "Password" with "SecurePass123"
    And I press "Create Account"
    And wait for AJAX to finish
    Then I should see "Welcome!"

    When I press "Take the Tour"
    And wait for AJAX to finish
    Then I should see "Step 1 of 3"

    When I press "Next"
    Then I should see "Step 2 of 3"

    When I press "Next"
    Then I should see "Step 3 of 3"

    When I press "Finish"
    Then I should see "You are all set!"
    And I should see "Dashboard"
```

Yes, this scenario asserts seven times, a deliberate exception to “one behavior per scenario”. The behavior under test here is the journey itself: onboarding is a sequence, and a user stranded at step 2 of 3 is the failure mode that matters. Use this pattern sparingly (one or two journey scenarios per product, tagged `@slow`), and keep each individual screen’s edge cases in their own small scenarios. When a journey scenario fails, the failing step tells you which leg of the trip broke, that is the pattern earning its length.

Pattern: The Constraint Scenario

Test what should NOT happen:

```
Feature: Security Constraints
  Scenario: Non-admin cannot access admin pages
    Given I am a logged in user with the "viewer" user
    When I go to "/admin/settings"
    Then I should see "Access Denied"
    And I should not see "System Settings"

  Scenario: Logged out user cannot access dashboard
    Given I am on homepage
    When I go to "/dashboard"
    Then I should not see "Dashboard"
    And I should see "Login"
```

Negative assertions alone are weak evidence, “I should not see ‘Dashboard’” also passes on a blank error page, a 404, or a half-rendered shell. That is why each scenario pairs the negative with a positive: “Access Denied” or “Login” proves the application responded with the page you expected, and THEN the absence check means something. Make the pair a habit; constraint scenarios are the ones auditors and security reviews read first, and they should hold up.

Pattern: The Before-After Scenario

Verify state change:

```
Feature: Profile Update
  Scenario: User updates their display name
    Given I am a logged in user with the "admin" user

    # Before: verify current state
    When I go to "/profile"
    Then I should see "admin"

    # Action: make the change
    When I go to "/profile/edit"
    And I fill in "Display Name" with "Super Admin"
    And I press "Save"
    And wait for AJAX to finish

    # After: verify new state
    When I go to "/profile"
    Then I should see "Super Admin"
```

The before-check earns its two extra steps by ruling out a false pass: without it, “I should see ‘Super Admin’” could succeed because the display name was ALREADY “Super Admin”, leftover state from manual testing on a shared environment, a seeded fixture, a previous run that did not clean up. Verifying the starting state first means the final assertion proves a transition, not a coincidence. Reach for this pattern whenever the suite runs against environments you do not fully reset between runs, which, on shared staging servers, is most of them.

Chapter Summary

- Gherkin has a small keyword set: Feature, Scenario, Given/When/Then, Background, Scenario Outline, Examples, Tags
- Background runs before every scenario, keep it short (4-5 lines max)
- Scenario Outline eliminates duplication with parameterized data tables
- Doc Strings handle multiline text (descriptions, JSON, HTML)
- Tags organize, filter, and control execution: use purpose, speed, environment, and domain tags
- Gherkin supports 70+ human languages for keywords
- Prefer declarative style (WHAT) over imperative (HOW), except when testing UI behavior
- Every scenario should tell a story that makes sense to non-technical readers

In the next chapter, we will build your first real test suite from scratch.

Chapter 4: Your First Real Test Suite

Theory is useful. Practice is essential. In this chapter, we build a complete test suite for a realistic web application. You will see the full BDD cycle in action: identify what to test, write the features, run them, and iterate.

4.1 Choosing What to Test First

When you start testing an existing application, the temptation is to test everything. Resist that temptation. Start small. Start with what matters most.

The Testing Pyramid

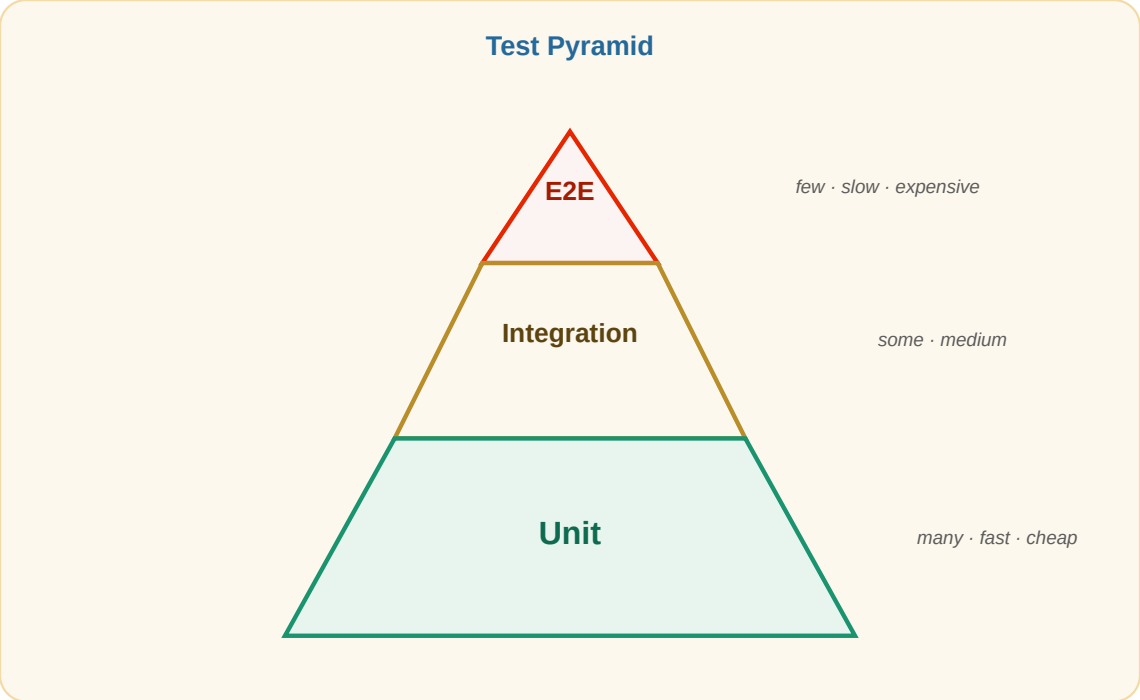


FIGURE 4.1 *The Testing Pyramid*

webship-js tests are at the top of the pyramid: end-to-end acceptance tests. They are slower than unit tests but they test the real user experience. Because they are slower, be strategic about what you test.

What to Test First (Priority Order)

RANK	WHAT TO TEST
1	Authentication (login, logout, session)
	→ If users cannot log in, nothing else matters
2	Critical business flows (checkout, signup, main feature)
	→ The flows that generate revenue or provide core value
3	Common user journeys (navigation, search, profile)
	→ The paths most users follow most of the time
4	Admin functionality (user management, settings)
	→ Important but used less frequently
5	Edge cases and error handling
	→ The things that break when you least expect it
6	API endpoints
	→ Backend validation independent of UI

Start with the Happy Path

For each feature, write the successful flow first. Then add failure scenarios. Then edge cases. This gives you the fastest coverage of the most important behaviors.

4.2 Building a Login Test Suite

Authentication is the gateway to your application. If login breaks, every authenticated feature breaks. Always test it first.

Recipe 21 Testing Successful Login

Problem: A Friday evening deploy subtly broke the session cookie, and nobody could log in until Monday morning when the support tickets arrived in a pile. Every authenticated feature was down all weekend, and every authenticated test would have caught it, if any had run. Login is the front door: when it breaks, the blast radius is the whole product. It deserves the first scenario you ever write for a project, and that scenario should run on every single deploy.

Solution:

Create `tests/features/authentication/login.feature`:

```
Feature: User Login
  As a registered user
  I want to log into my account
  So that I can access my personalized dashboard

  @smoke @auth @fast
  Scenario: Successful login with valid credentials
    Given I am on homepage
    When I go to "/login"
    Then I should see "Sign In"
    And I should see "Email"
    And I should see "Password"

    When I fill in "Email" with "testuser@example.com"
    And I fill in "Password" with "ValidPass123"
    And I press "Sign In"
    And wait for AJAX to finish
    Then I should see "Dashboard"
    And I should see "Welcome"
    And I should not see "Sign In"
```

How It Works:

This scenario tests the complete happy path:

1. Navigate to the login page
2. Verify the form exists (defensive check)
3. Fill in credentials
4. Submit the form
5. Verify the user lands on the dashboard
6. Verify the login form is no longer visible

The defensive check in the middle (`Then I should see "Sign In"`) catches a common problem: the test navigates to `/login` but the page did not load correctly. Without this check, the test would try to fill in fields that do not exist and give a confusing error. With it, a broken login PAGE fails at step three with "Sign In not found", while broken login LOGIC fails at the dashboard assertion, two different bugs, immediately distinguishable in the report.

A few things happen between the lines. The fill steps resolve "Email" and "Password" label-first, the visible label, then id, name, and placeholder, so the scenario survives markup changes that keep the labels. After `I press "Sign In"`, webship-js's auto-settle hook is already waiting out the form submission; the explicit `wait for AJAX to finish` adds a larger budget and, just as importantly, documents in plain text that a server round-trip happens here. And the closing trio matters as a set: "Dashboard" proves navigation, "Welcome" proves the page rendered user content, and `I should not see "Sign In"` proves the app did not silently bounce back to the form, the failure mode where login "works" but the session never sticks.

At scale: This scenario is deliberately shaped so it can be identical on every project you run, only the labels and landing-page text vary. Make `login.feature` the first file in your starter template, tag it `@smoke`, and wire `--tags @smoke` into every deploy pipeline. When login coverage is a template inherited on day one rather than a task on a backlog, the weekend-outage story above stops being possible anywhere in the portfolio.

Recipe 22 Testing Failed Login Attempts

Problem: The security review asks a simple question: "what does your login form do with bad input?", and the test suite has no answer, because everyone tested the door opening and nobody tested it staying shut. The expensive version of this gap is real: an application that said "wrong password" for existing accounts and "user not found" for everything else, quietly handing attackers a list of valid emails. Rejection behavior is a feature with its own requirements, and it needs its own scenarios.

Solution:

```
Feature: Login Failure Handling
  As a security-conscious application
  I want to reject invalid login attempts
  So that unauthorized users cannot access accounts

  Background:
    Given I am on homepage
    When I go to "/login"

  @auth @regression
  Scenario: Wrong password
    When I fill in "Email" with "testuser@example.com"
    And I fill in "Password" with "WrongPassword123"
    And I press "Sign In"
    And wait for AJAX to finish
    Then I should see "Invalid credentials"
    And I should not see "Dashboard"

  @auth @regression
  Scenario: Non-existent email
    When I fill in "Email" with "nobody@example.com"
    And I fill in "Password" with "AnyPassword123"
    And I press "Sign In"
    And wait for AJAX to finish
    Then I should see "Invalid credentials"
    And I should not see "Dashboard"

  @auth @regression
  Scenario: Empty email field
    When I fill in "Password" with "SomePassword"
    And I press "Sign In"
    Then I should see "Email is required"

  @auth @regression
  Scenario: Empty password field
    When I fill in "Email" with "testuser@example.com"
    And I press "Sign In"
    Then I should see "Password is required"
```

```
@auth @regression
Scenario: Both fields empty
  When I press "Sign In"
  Then I should see "Email is required"
```

How It Works:

Notice the Background section. Every scenario starts on the login page. We test five failure modes:

1. Correct email, wrong password
2. Non-existent email
3. Missing email
4. Missing password
5. Both missing

Also notice: the error message is generic ("Invalid credentials") for wrong password AND non-existent email. This is a security best practice, do not reveal whether the email exists in your system. The two scenarios assert the SAME message on purpose; the day a developer "improves" the error to be more specific, this pair fails and forces the conversation about user enumeration before the change ships.

Look at which scenarios wait and which do not. The first two failure modes submit to the server, so they keep `wait for AJAX to finish`; the empty-field scenarios assert immediately, because required-field validation should fire client-side without a round-trip. That difference is itself part of the spec, if "Email is required" suddenly needs an AJAX wait to pass, client-side validation broke and users are now paying a server round-trip to learn they forgot a field. The Background, meanwhile, is just the navigation; each scenario still runs in a fresh browser context, so a failed attempt in one scenario cannot trip a rate-limiter or leave error state for the next.

At scale: These five failure modes are a portfolio baseline, not a per-project invention. Put the five-scenario shape in your starter template next to the happy path from Recipe 21, and reviews become mechanical: which of the five is missing, and was that deliberate? When the security team audits 20 products, "every login form has the same five rejection scenarios, and the enumeration check is one of them" is an answer that ends the meeting early.

Recipe 23 Testing Password Reset Flow

Problem: Password reset is the feature nobody exercises for months, until a user is locked out, already frustrated, and discovers the reset link errors out because a refactor three releases ago broke a flow no test was watching. It is the classic silent-rot feature: rarely used, emotionally critical when used, and spanning a boundary (the email system) that makes people throw up their hands and not test it at all. You can test most of it today, and you should.

Solution:

```
Feature: Password Reset
  As a user who forgot their password
  I want to reset my password via email
  So that I can regain access to my account

  @auth @regression
  Scenario: Request password reset
    Given I am on homepage
    When I go to "/login"
    And I click "Forgot Password?"
    Then I should see "Reset Your Password"

    When I fill in "Email" with "testuser@example.com"
    And I press "Send Reset Link"
    And wait for AJAX to finish
    Then I should see "Check your email"
    And I should see "We have sent a password reset link"

  @auth @regression
  Scenario: Password reset with non-existent email
    Given I am on homepage
    When I go to "/login"
    And I click "Forgot Password?"
    And I fill in "Email" with "unknown@example.com"
    And I press "Send Reset Link"
    And wait for AJAX to finish
    # Security: same message whether email exists or not
    Then I should see "Check your email"

  @auth @regression
  Scenario: Password reset with invalid email format
    Given I am on homepage
    When I go to "/login"
    And I click "Forgot Password?"
    And I fill in "Email" with "not-an-email"
    And I press "Send Reset Link"
    Then I should see "Please enter a valid email"
```

How It Works:

Be precise about what these scenarios prove and what they do not. They verify the application's side of the contract: the link exists on the login page, the request form works, and the app responds with its promise ("Check your email"). They do NOT verify that an email actually arrived, that requires pointing the test at a mail-catcher or checking via API, which Chapter 11's API steps make possible once you have one. Testing the reachable 80% today beats testing nothing while waiting for the perfect email harness.

The second scenario is the quiet star of this feature file. Same "Check your email" message for an address that does not exist, the comment in the scenario says why, and the scenario enforces it. Reset forms are the OTHER classic user-enumeration leak (after login errors, Recipe 22): a form that answers "no account with that email" is a free account-discovery oracle. Note also that the invalid-format scenario asserts without an AJAX wait, expecting client-side validation, while the two server-bound scenarios wait, the same client/server split you saw in Recipe 22, applied consistently.

Recipe 24 Testing Session Persistence

Problem: The SSO migration shipped on Tuesday, and by Wednesday support was drowning: every page refresh logged users out, because the new token lived in JavaScript memory instead of a cookie. Login itself tested green, you could sign in beautifully, once, until you touched F5. "Logged in" and "stays logged in" are different promises kept by different mechanisms, and a suite that only tests the first will wave the second's regressions straight through.

Solution:

```
Feature: Session Management
  As a logged-in user
  I want my session to persist as I navigate
  So that I do not need to log in on every page

  @auth @smoke
  Scenario: Session persists across pages
    Given I am a logged in user with the "admin" user
    When I go to "/dashboard"
    Then I should see "Dashboard"

    When I go to "/profile"
    Then I should see "Profile"
    And I should not see "Sign In"

    When I go to "/settings"
    Then I should see "Settings"
    And I should not see "Sign In"

  @auth @regression
  Scenario: Session persists after page reload
    Given I am a logged in user with the "admin" user
    When I go to "/dashboard"
    Then I should see "Dashboard"

    When I reload the page
    Then I should see "Dashboard"
    And I should not see "Sign In"
```

How It Works:

The two scenarios attack the two ways sessions die. Navigation between pages tests that the session credential travels with ordinary requests, a cookie scoped to the wrong path, or a token attached only by one page's JavaScript, fails here. `When I reload the page` tests the harder case: a full reload tears down all in-memory JavaScript state, so anything not persisted in cookies or storage is gone. The in-memory-token bug from the problem statement passes scenario one and fails scenario two, which is exactly why both exist. Within a scenario, webship-js keeps one browser context alive, so cookies and localStorage persist across these navigations just as they would for a real user; between scenarios, the context is rebuilt from scratch, so each scenario proves persistence independently rather than borrowing the previous one's session.

There is a second reason to keep this scenario green: webship-js's auth-state steps, the "log in once, save the auth state, restore it everywhere" pattern, work by snapshotting and re-injecting exactly this persisted state (cookies plus storage). If sessions do not survive a reload, saved auth state will not survive a restore either. This scenario is the canary for that whole optimization.

At scale: That makes this small feature file pull double duty across a portfolio. Once your projects adopt auth-state reuse to cut CI time (and they should, it is usually the biggest runtime win available, see Recipe 4's numbers), every suite silently depends on session persistence working. Keep this scenario tagged `@smoke` on every project: when an SSO or cookie-policy change breaks persistence, you want the named, two-scenario canary failing loudly, not forty unrelated scenarios failing mysteriously because their restored sessions evaporated.

4.3 Building a Registration Test Suite

Recipe 25 Testing Complete Registration

Problem: Marketing launches the spring campaign on Monday, traffic triples, and Tuesday's standup learns that the redesigned signup page has been silently rejecting everyone whose browser autofilled the Terms checkbox differently. Registration is the revenue front door, every broken hour is measurable in lost accounts, and yet it is usually tested less than login, because "we would notice". You notice faster with a scenario that walks the whole form on every deploy.

Solution:

```
Feature: User Registration
  As a new visitor
  I want to create an account
  So that I can access member features

  @auth @smoke
  Scenario: Successful registration
    Given I am on homepage
    When I go to "/register"
    And I fill in "First Name" with "Test"
    And I fill in "Last Name" with "User"
    And I fill in "Email" with "newuser-test@example.com"
    And I fill in "Password" with "SecurePass123!"
    And I fill in "Confirm Password" with "SecurePass123!"
    And I check "I agree to the Terms of Service"
    And I press "Create Account"
    And wait for AJAX to finish
    Then I should see "Welcome"
    And I should see "Your account has been created"
```

How It Works:

Every input type on the form gets exercised through its label. The text fields go through the usual label-first resolution, and `I check "I agree to the Terms of Service"` does the same for the checkbox, it finds the checkbox by its visible label text and checks it, which matters because terms checkboxes are exactly the elements designers love to restyle into custom widgets where the real `<input>` is hidden. If a redesign breaks the label-to-input association (the autofill bug in the problem statement was one of these), this step fails with a clear "cannot find" error instead of a silent un-checked box. The final two assertions are deliberately both present: "Welcome" alone could be the homepage greeting; "Your account has been created" pins the success to THIS action.

The trap in this recipe is the email address. `newuser-test@example.com` registers fine on the first run, and on the second run the same address collides with the account the first run created, and your registration test fails for being too successful. Decide on day one how the suite gets a fresh identity each run: a test-data reset job before the suite, an environment that wipes between nightly runs, or a custom step that generates a unique address (Chapter 17 territory). Any of the three works; having none of them means a registration suite that only passes on Mondays.

At scale: Test-data hygiene is a portfolio policy, not a per-suite hack. Pick ONE of the three strategies above and make it the standard, mixed strategies mean the person debugging project N first has to discover which trick project N uses. The reset-before-run approach is usually the right default for staging environments because it also serves Recipe 27, which needs a known account to already exist: one seeding script can guarantee both "this email is free" and "that email is taken".

Recipe 26 Testing Field Validation with Scenario Outline

Problem: The registration form's validation rules live in a spreadsheet the backend developer keeps locally, and the only executable record of them is scattered through nine copy-pasted scenarios that differ by two words each. When the password policy changes from 8 characters to 10, someone updates the application, the spreadsheet, and five of the nine scenarios. The other four keep testing the old rule, passing for the wrong reasons. Validation rules are tabular by nature; the tests should be tabular too, with exactly one place to edit.

Solution:

```
Feature: Registration Validation
  As the application
  I want to validate registration fields
  So that only valid data enters the system

  Scenario Outline: Field validation rules
    Given I am on homepage
    When I go to "/register"
    And I fill in "<field>" with "<value>"
    And I press "Create Account"
    Then I should see "<error>"

    Examples: Email validation
      | field  | value             | error                          |
      | Email  | not-an-email      | Please enter a valid email     |
      | Email  | @missing-local    | Please enter a valid email     |
      | Email  | spaces in@mail.co | Please enter a valid email     |

    Examples: Password validation
      | field    | value    | error                                |
      | Password | 123      | Password must be at least 8 chars    |
      | Password | password | Password must include a number       |
      | Password | 12345678 | Password must include a letter       |

    Examples: Name validation
      | field      | value | error                      |
      | First Name |       | First name is required     |
      | Last Name  |       | Last name is required      |
```

This generates **9 test scenarios** from a single template, covering email, password, and name validation rules.

How It Works:

Each Examples row becomes a complete, independent scenario, fresh browser context, fresh navigation to `/register`, one field filled, one submit, one assertion. The labeled Examples blocks are not decoration: when the run reports a failure, it names the block and the row, so "Password validation, row `| password | Password must include a number |`" tells you which rule regressed before you open anything. Note there is no `wait for AJAX to finish` anywhere in the template, these rules should fire client-side, and the outline quietly enforces that expectation for all nine cases at once.

Mind the cost model: nine rows means nine browser sessions, which is the right price for nine real business rules. What the outline must not become is a fuzzer, if you want to throw 200 malformed emails at the validator, do that in a unit test against the validation function, and keep the acceptance rows to the cases a product owner would recognize as policy.

At scale: This table IS the validation spec, in a format a product owner can edit. When the password policy changes, the policy change arrives as a PR that edits one row, visible, reviewable, and impossible to half-apply the way the nine scattered scenarios were. Share the outline's shape across projects and the spreadsheet-on-someone's-laptop problem disappears portfolio-wide: the rules live in the repo, and the nightly run notarizes them.

Recipe 27 Testing Duplicate Email Prevention

Problem: Support finds a customer with three accounts under one email, three carts, three loyalty balances, and a password reset that updates a different account than the one she logs into. The uniqueness check existed once, then an auth refactor moved account creation to a new endpoint and the check did not move with it. No single feature "looked" broken, which is why this regression class ships so easily: only a test that tries the duplicate, on every deploy, keeps the constraint honest.

Solution:

```
Feature: Duplicate Email Prevention
  As the application
  I want to prevent duplicate registrations
  So that each email has only one account

  @auth @regression
  Scenario: Registration with already-used email
    Given I am on homepage
    When I go to "/register"
    And I fill in "First Name" with "Another"
    And I fill in "Last Name" with "User"
    And I fill in "Email" with "existing@example.com"
    And I fill in "Password" with "SecurePass123!"
    And I fill in "Confirm Password" with "SecurePass123!"
    And I check "I agree to the Terms of Service"
    And I press "Create Account"
    And wait for AJAX to finish
    Then I should see "An account with this email already exists"
    And I should not see "Welcome"
```

How It Works:

This scenario has a hidden dependency, and it is healthier to say so out loud: `existing@example.com` must already have an account in the environment under test, or the registration succeeds and the scenario fails for the wrong reason. That makes this a fixture-dependent test. The clean way to satisfy it is the seeding script from Recipe 25's At-scale note, the same job that frees up the fresh address guarantees the taken one. The self-contained alternative is a scenario that registers an email and then immediately tries to register it again; it removes the fixture dependency at the cost of a longer scenario and a per-run unique address. Either is fine. What is not fine is the implicit version, where the test passes only because someone manually created that account on staging in 2024.

The closing pair of assertions mirrors the constraint pattern from Chapter 3: the positive ("already exists" message appeared) proves the application responded as designed, and the negative ("Welcome" absent) proves no second account slipped through behind a friendly-looking error.

One last design question worth a Three Amigos minute: should THIS error be specific while login errors stay generic? Most teams accept the enumeration trade-off here, a registration form necessarily reveals taken emails, but it should be a recorded decision, not an accident.

At scale: Fixture data is an environment contract, and it belongs in version control next to the suite, a seed script or documented fixture list that says “these accounts exist, with these roles, on every test environment”. When 20 projects each assume slightly different invisible staging state, environment rebuilds become archaeology. When the assumptions are a checked-in script, rebuilding staging for any project is one command, and this scenario passes on a fresh environment the same day.

4.4 The Red-Green-Refactor Cycle in Practice

Let us walk through the full BDD cycle for a single scenario.

Step 1: Write the Scenario (RED)

```
Feature: User Profile
  Scenario: User views their profile
    Given I am a logged in user with the "admin" user
    When I go to "/profile"
    Then I should see "My Profile"
    And I should see "admin"
```

Step 2: Run It

```
npm test
```

Output (if the feature does not exist yet):

```
1 scenario (1 failed)
4 steps (2 passed, 1 failed, 1 skipped)

Failing step:
  When I go to "/profile"
  Error: Navigation to http://localhost:8080/profile failed: 404
```

This is RED. The `/profile` page does not exist yet. That is expected and correct.

Step 3: Implement (GREEN)

Build the `/profile` page in your application. Show the user's name. Then run the test again:

```
npm test
```

```
1 scenario (1 passed)
4 steps (4 passed)
```

GREEN. The scenario passes.

Step 4: Refactor

Now improve the code. Maybe extract a profile component, clean up the routing, improve the styling. Run the tests after each change:

```
npm test
```

If the tests still pass, your refactoring did not break anything. If a test fails, your refactoring changed behavior, fix it.

The Safety Net

This is the power of BDD. Every time you refactor, the test suite tells you immediately if you broke something. You can refactor with confidence because you have a safety net of executable specifications.

4.5 Running and Interpreting Results

Recipe 28 Understanding Cucumber Output States

Problem: A teammate pastes a wall of dots and one red `F` into the team chat with "tests are broken, help." Before anyone can help, they have to know what the symbols mean, and the difference between a `F`, a `U`, and an `A` is the difference between a real bug, a step you forgot to write, and two step definitions fighting over the same sentence. Until the whole team reads these six states the same way, every failure turns into a guessing game.

Solution:

STATE	SYMBOL	WHAT IT MEANS
Passed	.	Step executed successfully
	(green)	The assertion was met
Failed	F	Step executed but assertion failed
	(red)	OR an error/exception occurred
		Remaining steps in scenario are **SKIPPED**
Undefined	U	No step definition matched this step text
	(yellow)	Cucumber will suggest a snippet to use
Pending	P	Step definition exists but is incomplete
	(yellow)	Contains 'return "pending"' or similar
Skipped	-	Not executed because a previous step failed
	(cyan)	System in unknown state, unsafe to continue
Ambiguous	A	Multiple step definitions match this text
	(red)	Fix: make step patterns more specific

How It Works:

The two states people confuse most are Failed and Undefined, and they point at opposite problems. A red `F` means the step ran and the application did the wrong thing, a genuine signal worth waking someone for. A yellow `U` means Cucumber never found a step definition matching that sentence, so nothing about the application was tested at all; the suite is reporting on a gap in your test code, not a gap in the product. Treating a `U` like an `F` sends a developer hunting through application code for a bug that lives in the feature file.

Skipped (`-`) is the one that protects you. The moment a step fails, Cucumber abandons the rest of that scenario, because once an action did not happen the page is in an unknown state and every later assertion would be lying. So a scenario reports one real failure followed by a trail of cyan skips, the skips are not extra problems, they are Cucumber refusing to guess. Ambiguous (`A`) almost always means two people wrote a step definition for the same phrase in different files; with webship-js loading every file under `tests/step-definitions/**/*.js` into one global registry, the fix is to make one pattern more specific or delete the duplicate.

At scale: When twenty projects share one chat channel, agree on what each state obligates. A red `F` pages the on-call engineer; a `U` or `A` is a test-code defect that goes to whoever wrote the feature, never to the product team; a wall of skips after one failure is one bug, not twenty. Writing that convention down once, and pointing every new hire at this table, stops the daily “is this real?” triage tax that otherwise scales linearly with the number of suites you run.

Recipe 29 Using Different Reporters

Problem: The format that helps a developer watch a local run, one line per step, scrolling past, is useless to a CI server that just needs a compact pass/fail stream, and worse than useless to a release manager who wants a link they can open in a browser. One run, three audiences, and if you only know one reporter you end up serving all three badly.

Solution:

```
# Progress bar (default - great for CI)
npx cucumber-js --format progress-bar

# Detailed output (shows each step)
npx cucumber-js --format progress

# Summary only (just pass/fail counts)
npx cucumber-js --format summary

# JSON output (for reporting tools)
npx cucumber-js --format json:reports/results.json

# HTML report (visual, shareable)
npx cucumber-js --format html:reports/report.html

# Multiple formats at once
npx cucumber-js \
  --format progress-bar \
  --format json:reports/results.json \
  --format html:reports/report.html
```

How It Works:

Reporters are independent consumers of the same run, you are not re-running tests to get a second format, you are attaching another listener to the one execution. That is why you can stack `--format` flags: the progress bar prints to your terminal for a human watching, while `json:reports/results.json` writes the machine-readable record to disk in the same pass. The JSON file is the important one for teams. It is the canonical artifact webship-js's own report generator (`bin/generate-reports.js`) reads to build the HTML and PDF, and it carries every step, status, timing, and attached screenshot, so it is the single source everything downstream is derived from.

The default `progress-bar` is deliberately terse because CI logs are read by machines and skimmed by humans, one character per step keeps a thousand-step run to a few screens. The verbose `progress` format is for the moment you are debugging a specific scenario locally and want to see each step's text scroll by.

At scale: Standardize on writing the JSON report at a fixed path in every project, even the ones nobody has wired a dashboard to yet. The terminal format is a personal preference that can vary by developer; the JSON artifact is a contract. Once every suite across the portfolio emits `reports/results.json` in the same place, one CI template can collect them, one script can roll them into a cross-project quality dashboard, and a new project joins the dashboard for free the day it lands, no per-project reporting code to write.

Recipe 30 Generating HTML Reports

Problem: Your suite is green, but the delivery manager three time zones away cannot run `npm test` and would not read terminal output if they could. They keep asking "is the release safe?" in standups, and "trust me, the tests pass" stops being a satisfying answer around the third release. They need something they can open in a browser and read themselves.

Solution:

Add HTML reporting to your `cucumber.js`:

```
module.exports = {
  default: {
    require: ['node_modules/webship-js/tests/step-definitions/*.js'],
    paths: ['tests/features/**/*.feature'],
    worldParameters: { launchUrl: 'http://localhost:8080' },
    format: [
      'progress-bar',
      'html:reports/cucumber-report.html'
    ],
    publishQuiet: true
  }
};
```

Run tests:

```
npm test
```

Open the report:

```
open reports/cucumber-report.html
# or on Linux:
xdg-open reports/cucumber-report.html
```

The HTML report shows:

- Total scenarios passed/failed
- Each feature with its scenarios
- Each step with pass/fail status
- Error messages and stack traces for failures
- Execution time per scenario

How It Works:

The built-in `html` formatter is the quick option, Cucumber writes a self-contained page as the run finishes, no extra step. For a report you actually hand to a stakeholder, webship-js ships `bin/generate-reports.js`, which reads the JSON artifact from Recipe 29 and produces a richer HTML or PDF document with selectable themes and a metadata header. Crucially, the screenshots webship-js captures on failure (it sets `WEBSHIP_SCREENSHOT_ON_FAILED` true by default) are embedded right into that report, so a failure shows the picture of the broken page next to the error, a non-engineer can see what went wrong without reading a stack trace. That turns the report from a pass/fail log into evidence.

At scale: A green checkmark in CI tells a manager the run finished; an HTML or PDF report tells them *what was verified*. Stamp each report with `--app-version`, `--env`, and `--browser` (the metadata flags `generate-reports.js` accepts) so a release manager comparing two builds, or an auditor six months later asking “what did we test for the March release?”, reads the answer straight off the document. When every project emits the same report shape, the question stops being “do you trust QA?” and becomes “open the report”, which is exactly the cross-project, manager-readable quality signal a PMO needs and the kind of self-serve evidence that ends the standup interrogation.

4.6 Your First Test Suite Checklist

Before you call your initial test suite “done,” make sure you have covered:

TEST SUITE CHECKLIST

- ☐ Project Setup
  - ☐ `package.json with test scripts`
  - ☐ `cucumber.js with correct launchUrl`
  - ☐ `Directory structure (features/, step-definitions/, etc)`
  - ☐ `.gitignore (node_modules/, reports/)`
- ☐ Smoke Tests (run on every deployment)
  - ☐ Homepage loads
  - ☐ Login works
  - ☐ Main navigation works
  - ☐ Critical feature accessible
- ☐ Authentication Tests
  - ☐ Successful login
  - ☐ Failed login (wrong password)
  - ☐ Failed login (non-existent user)
  - ☐ Empty field validation
  - ☐ Logout
  - ☐ Session persistence
- ☐ Core Feature Tests
  - ☐ Happy path for the main feature
  - ☐ At least one failure scenario
  - ☐ At least one edge case
- ☐ Organization
  - ☐ Features organized by domain
  - ☐ `Tags applied (@smoke, @regression, @auth, etc.)`
  - ☐ Background used for shared setup
  - ☐ Scenario Outline used for data-driven tests
- ☐ CI Integration
  - ☐ Tests run in CI pipeline
  - ☐ Reports generated
  - ☐ Build fails on test failure

FIGURE 4.2 *Your First Test Suite Checklist*

A Real-World Example: Blog Application Test Suite

Let me show you what a complete test suite looks like for a blog application. This ties together everything from Chapters 1-4.

File Structure

```
blog-tests/
├── cucumber.js
├── package.json
├── tests/
│   ├── features/
│   │   ├── smoke/
│   │   │   └── health-check.feature
│   │   ├── authentication/
│   │   │   ├── login.feature
│   │   │   └── registration.feature
│   │   ├── blog/
│   │   │   ├── create-post.feature
│   │   │   ├── edit-post.feature
│   │   │   ├── view-posts.feature
│   │   │   └── comments.feature
│   │   └── admin/
│   │       └── user-management.feature
│   └── assets/
│       └── test-image.png
└── reports/
```

FIGURE 4.3 *File Structure*

health-check.feature

```
@smoke @fast
Feature: Health Check
  As a DevOps engineer
  I want to verify the site is up
  So that I know deployments succeeded

  Scenario: Homepage loads
    Given I am on homepage
    Then I should see "Blog"

  Scenario: Login page loads
    Given I am on homepage
    When I go to "/login"
    Then I should see "Sign In"
```

create-post.feature

```
@javascript @blog
Feature: Create Blog Post
  As a blog author
  I want to create new blog posts
  So that I can share my thoughts with readers

  Background:
    Given I am a logged in user with the "editor" user
    And I go to "/admin/posts/new"
    And wait for AJAX to finish

  @smoke
  Scenario: Create a simple blog post
    When I fill in "Title" with "Getting Started with BDD"
    And I fill the rich text editor field "Body" with "<p>BDD is a
powerful approach to software development.</p>"
    And I press "Publish"
    And wait for AJAX to finish
```

```
    Then I should see "Post published successfully"

  @regression
  Scenario: Create post with all fields
    When I fill in "Title" with "Complete Guide to webship-js"
    And I fill the rich text editor field "Body" with "<p>Learn automated
testing with webship-js.</p>"
    And I fill in "Excerpt" with "A comprehensive guide"
    And I select "Technology" from "Category"
    And I fill in "Tags" with "bdd, testing, automation"
    And I attach the file "test-image.png" to "Featured Image"
    And I press "Publish"
    And wait for AJAX to finish
    Then I should see "Post published successfully"

  @regression
  Scenario: Cannot create post without title
    And I fill the rich text editor field "Body" with "<p>Some
content</p>"
    And I press "Publish"
    Then I should see "Title is required"

  @regression
  Scenario: Create post as draft
    When I fill in "Title" with "Draft Post"
    And I fill the rich text editor field "Body" with "<p>Work in
progress.</p>"
    And I press "Save as Draft"
    And wait for AJAX to finish
    Then I should see "Draft saved"
```

Running the Suite

```
# Run everything
npm test

# Run just smoke tests
npm run test:smoke

# Run just blog tests
npx cucumber-js --tags "@blog"

# Run blog tests excluding WIP
npx cucumber-js --tags "@blog and not @wip"
```

Chapter Summary

- Start testing with authentication (login/logout), then critical business flows
- Follow the testing pyramid: E2E tests (webship-js) at the top, focused on features
- Begin with the happy path, then add failure and edge case scenarios
- The Red-Green-Refactor cycle works at the feature level: write scenario → fail → implement → pass → refactor
- Use reporters (progress-bar, HTML, JSON) for different audiences
- A production-ready test suite has smoke tests, auth tests, core feature tests, and CI integration
- Organize features by domain, use tags for filtering, Background for shared setup

Part I is complete. You now understand BDD, have webship-js set up, know Gherkin inside out, and have built a real test suite. In Part II, we dive into the recipes, specific solutions for every type of web testing challenge.

Part II: Browser Testing Recipes

Chapter 5: Navigation Recipes

Navigation is the foundation of every web test. Before you can test a form, a modal, or an API response, you need to get to the right page. This chapter covers every navigation pattern you will encounter.

Recipe 31 Navigating to the Homepage

Problem: It is the first day on a new project, and the first thing someone does is hardcode `http://localhost:8080` into a feature file. It works on their laptop. Three weeks later the same suite needs to run against staging in CI, and now there are URLs sprinkled through forty scenarios that all need changing. Every test needs a starting point, and that starting point needs to live in exactly one place.

Solution:

```
Feature: Homepage Navigation
  Scenario: Visit the homepage
    Given I am on homepage
    Then I should see "Welcome"
```

How It Works:

`Given I am on homepage` navigates to the URL configured as `launchUrl` in your `cucumber.js` worldParameters. The `LAUNCH_URL` environment variable overrides it, and the default is `http://localhost:8080`. Your feature files never mention a domain, the environment decides.

Behind the scenes, the step definition does roughly this:

```
// What webship-js does internally (simplified)
await this.page.goto(this.launchUrl);
await smartSettle(this.page, 3000); // per-navigation settle budget
```

The interesting part is `smartSettle`. webship-js installs three window globals on every page via `addInitScript`: `window.__webshipAjaxCount` (a fetch/XHR wrapper that counts in-flight requests), `window.__webshipPendingTimers` (a `setTimeout` wrapper), and `window.__webshipLastMutation` (a MutationObserver timestamp). After the body is attached, webship-js probes `domcontentloaded` and `networkidle`, then polls those three counters every 50ms. The page counts as settled when AJAX is at zero, no timers are pending, and the DOM has been quiet for at least 250ms. That is why there are no static sleeps anywhere in webship-js, the framework waits for evidence, not for time.

At scale: When one team runs the same suite against twenty sites, `LAUNCH_URL` is the whole trick. The CI job for each project exports its own value, `LAUNCH_URL=https://staging.project-a.example`, and the feature files are byte-for-byte identical across projects. The moment a base URL appears inside a `.feature` file instead of an environment variable, that suite stops being shareable. Treat it like a code review rule.

Recipe 32 Navigating to Specific Pages

Problem: Your suite needs to visit the about page, the product catalog, the contact form, a dozen distinct pages per feature. Click-through navigation from the homepage works, but it is slow and it couples every test to the menu structure: redesign the header and thirty scenarios break before they reach the thing they actually test. You want to land directly on the page under test.

Solution:

```
Feature: Page Navigation
  Scenario: Navigate to specific pages
    Given I am on homepage

    When I go to "/about"
    Then I should see "About Us"

    When I go to "/products"
    Then I should see "Our Products"

    When I go to "/contact"
    Then I should see "Contact Us"

    When I go to "/blog"
    Then I should see "Latest Posts"
```

How It Works:

`When I go to "/about"` appends the path to your `launchUrl`. If your base URL is `http://localhost:8080`, this navigates to `http://localhost:8080/about`.

The path must start with `/`. webship-js concatenates:

```
launchUrl + path = final URL
http://localhost:8080 + /about = http://localhost:8080/about
```

Each navigation gets the same treatment as the homepage step: `page.goto()` followed by `smartSettle` with its 3000ms per-navigation budget, watching `window.__webshipAjaxCount`, pending timers, and DOM mutations until the page is genuinely quiet. Direct navigation also keeps each scenario testing one behaviour, if you need five clicks to reach the page, the clicks become five extra ways for the scenario to fail for reasons that have nothing to do with what it asserts. Navigate straight to the page; test the journey separately, once, in its own scenario.

Recipe 33 Navigating to Full URLs

Problem: Not everything lives under one domain. The docs are on `docs.example.com`, login goes through an SSO provider on another host, and the marketing site links out to a partner page. The day someone moves the documentation and forgets to update the links, you want a test that notices, which means your suite occasionally has to leave the comfort of `launchUrl` and visit a full URL.

Solution:

```
Feature: External Navigation
  Scenario: Navigate to a full URL
    Given I am on "https://example.com"
    Then I should see "Example Domain"

  Scenario: Navigate between domains
    Given I am on homepage
    When I go to "/login"
    Then I should see "Sign In"

    # Navigate to a different application
    Given I am on "https://docs.example.com"
    Then I should see "Documentation"
```

How It Works:

When you provide a full URL (starting with `http://` or `https://`), webship-js navigates directly to that URL instead of appending to the `launchUrl`. The smart-wait machinery still applies, the `addInitScript` counters are installed per browser context, not per domain, so AJAX and timer tracking keeps working after the domain hop.

One detail that saves staging environments: webship-js ships with `ignoreHTTPSErrors: true` in `playwright.config.ts`, so a full URL pointing at a staging server with a self-signed certificate will not kill the test the way it would scare a browser user.

Use this for:

- Testing links to external documentation
- Multi-application test suites
- Testing production URLs from staging tests

At scale: Hardcoded cross-domain URLs are the second most common reason a shared suite stops being portable (the first is hardcoded `launchUrl`). If several projects share a docs site or an SSO host, put those hosts in environment variables too and keep only genuinely fixed third-party URLs literal in the feature file. And be deliberate about external checks in CI: a partner site having a bad day should not turn your release pipeline red, so tag external-dependency scenarios (for example `@external`) and run them in a separate, non-blocking job.

Recipe 34 Browser Back and Forward

Problem: A customer files a bug: “I pressed Back and the page broke.” Nobody on the team can reproduce it at first, because nobody tests the back button, we all navigate forward. In single-page applications especially, client-side routers are notorious for mishandling history: back lands on a blank view, forward double-fires a data fetch, the URL changes but the content does not. If your app has client-side routing, history navigation deserves its own scenario.

Solution:

```
Feature: Browser History
  As a user
  I want to use browser back/forward buttons
  So that I can navigate my browsing history

  Scenario: Navigate back and forward
    Given I am on homepage

    # Build up browsing history
    When I go to "/products"
    Then I should see "Products"

    When I go to "/products/laptop"
    Then I should see "Laptop"

    # Go back
    When I move backward one page
    Then I should see "Products"

    # Go back again
    When I move backward one page
    Then I should see "Welcome"

    # Go forward
    When I move forward one page
    Then I should see "Products"

    # Go forward again
    When I move forward one page
    Then I should see "Laptop"
```

How It Works:

- `When I move backward one page` calls `page.goBack()` in Playwright
- `When I move forward one page` calls `page.goForward()`

In a classic multi-page site, each of these triggers a real navigation and gets the full per-navigation `smartSettle` budget. In an SPA, the router intercepts the history event and swaps content client-side, no page load at all. This is where webship-js's AfterStep hook earns its keep: after every state-changing step it runs `smartSettle(page, 1500)`, so the AJAX the route change kicked off (counted by `window.__webshipAjaxCount`) and the rendering timers behind it are usually settled before your next `Then I should see` runs.

A common bug in SPAs is that the back button does not work correctly, the URL rewinds but the view does not, or state from the "future" page leaks backward. This recipe catches both, because it asserts on visible content at every hop, not just on the URL.

Pro Tip: Browser history navigation can trigger page loads or just URL changes (in SPAs). Add `And wait for AJAX to finish` after navigation if your app loads content dynamically on route changes, it is redundant when the AfterStep settle already caught everything, but it documents intent and costs nothing when the page is already quiet.

Recipe 35 Page Reload

Problem: A user spends ten minutes filling in their profile, hits Save, sees the success message, refreshes the page out of habit, and everything is back to defaults. The save wrote to memory, not to the server. Bugs like this slip through because manual testers rarely refresh after saving, and automated suites that never reload the page cannot see them either. Reload is the cheapest persistence test you will ever write.

Solution:

```
Feature: Page Reload Behavior
  Scenario: Data persists after reload
    Given I am a logged in user with the "admin" user
    When I go to "/dashboard"
    Then I should see "Dashboard"

    When I reload the page
    Then I should see "Dashboard"
    And I should not see "Sign In"

  Scenario: Form data handling on reload
    Given I am on homepage
    When I go to "/settings"
    And I fill in "Display Name" with "Rajab"
    And I press "Save"
    And wait for AJAX to finish
    Then I should see "Settings saved"

    When I reload the page
    And wait for AJAX to finish
    Then I should see "Rajab" value in the "Display Name" input element
```

How It Works:

`When I reload the page` calls `page.reload()` in Playwright. This is equivalent to pressing F5 in the browser. Because a reload is a fresh navigation, webship-js re-runs its `addInitScript` instrumentation, the AJAX counter, timer tracker, and mutation observer are reinstalled on the new document, and the full per-navigation `smartSettle` budget applies before the next step runs. Anything that lived only in JavaScript memory is gone after the reload; anything in cookies, localStorage, or the database survives. That boundary is exactly what this recipe tests:

- Session persistence (user stays logged in)
- Saved data appears after refresh
- Client-side state management
- Cache behavior

The second scenario is the one that catches real bugs: fill, save, reload, then assert the field still holds the value. If the success message appeared but the value vanished, you have found an optimistic UI that lied to the user.

Recipe 36 Verifying the Current URL

Problem: Login is supposed to land users on the dashboard, and unauthorized visitors are supposed to bounce to the sign-in page. Then one sprint someone changes the auth middleware, the redirect target silently becomes the homepage, and nobody notices until a customer asks why login "forgets where I was going". Redirects are invisible in code review, the only reliable way to catch a broken one is a test that follows it and checks where it ended up.

Solution:

```
Feature: URL Verification
  Scenario: Login redirects to dashboard
    Given I am on homepage
    When I go to "/login"
    And I fill in "Email" with "admin@example.com"
    And I fill in "Password" with "AdminPass123"
    And I press "Sign In"
    And wait for AJAX to finish
    Then I should see "Dashboard"

  Scenario: Unauthorized access redirects to login
    Given I am on homepage
    When I go to "/admin/settings"
    Then I should see "Sign In"
    And I should not see "Settings"
```

How It Works:

The scenarios above verify redirects by checking the **content** of the page you land on. If you see "Dashboard," you know you are on the dashboard page. If you see "Sign In," you know you were redirected to login.

When you also need to pin down the address itself, webship-js 2.0 has dedicated URL assertion steps:

```
Then I should be on "/dashboard"
Then the url should match "/orders/\d+"
```

`Then I should be on "/path"` compares the current path against what you expect, and `Then the url should match "regex"` lets you assert dynamic URLs, order IDs, slugs, pagination, with a pattern instead of an exact string.

So which should you use? Content first, URL second:

- URLs can change during redesigns; behaviour-level content assertions survive them
- Content verification catches more bugs (correct URL but wrong content rendered)
- It follows the BDD principle of testing behavior, not implementation

Reach for the URL steps when the URL **is** the behaviour: canonical redirects for SEO, locale prefixes like `/ar/`, or a payment provider that must return the user to an exact callback path.

At scale: Across many projects, the pair "content assertion + `Then I should be on`" is your best defence against the sneakiest redirect failure: the soft redirect to a styled error page. The content check alone can pass ("Welcome" appears on the error page header too); the URL check alone can pass (right address, blank body). Together they are cheap and nearly impossible to fool, which is why redirect scenarios make excellent always-on smoke tests in every project's pipeline.

Recipe 37 Redirect Testing

Problem: Your team restructures the blog URLs, new year, new information architecture. Six months of old links live in newsletters, social posts, and other people's bookmarks. The redirect map gets written once, during the migration, and then nobody touches it. The first time a redirect silently breaks is usually when an annoyed reader emails marketing. A handful of scenarios keeps the promise you made when you changed the URLs.

Solution:

```
Feature: URL Redirects
  As a site administrator
  I want old URLs to redirect to new ones
  So that bookmarks and external links still work

  Scenario: Old blog URL redirects to new structure
    Given I am on homepage
    When I go to "/blog/2024/my-old-post"
    And wait for AJAX to finish
    Then I should see "My Old Post"

  Scenario: HTTP redirects to HTTPS
    Given I am on homepage
    When I go to "/secure-page"
    Then I should see "Secure Content"
    And I should not see "Not Secure"

  Scenario: Trailing slash normalization
    Given I am on homepage
    When I go to "/about/"
    Then I should see "About Us"
```

How It Works:

Playwright's `page.goto()` follows HTTP redirect chains automatically, a 301 from `/blog/2024/my-old-post` to the new URL just happens, the same way it would in a real browser. Your step lands on the final destination, `smartSettle` waits for it to go quiet, and your assertion runs against the page the user would actually see. To pin the destination address as well as the content, add `Then I should be on "/new-path"` after the navigation, that catches the case where a redirect was changed to point at the wrong (but still rendering) page.

Pro Tip: Keep a single `redirects.feature` per project that mirrors your server's redirect map, one scenario per legacy URL family. When someone edits the redirect config, the matching feature change shows up in the same pull request, and when it does not, the red build asks the question for you.

Recipe 38 404 and Error Page Testing

Problem: Someone follows a dead link and gets a raw server stack trace, framework version, file paths, the works. Or worse: a blank white page with no way back. Error pages are the part of the site nobody designs twice and nobody tests at all, yet every user eventually sees one. You want proof that your 404 page is helpful, branded, and still has working navigation, on every deploy, not just the day it was built.

Solution:

```
Feature: Error Pages
  As a user
  I want to see helpful error pages
  So that I know what went wrong and how to fix it

  @regression
  Scenario: 404 page for non-existent URL
    Given I am on homepage
    When I go to "/this-page-does-not-exist-xyz"
    Then I should see "Page Not Found"
    And I should see "The page you are looking for does not exist"
    And I should see "Go to Homepage"

  @regression
  Scenario: 404 page still has navigation
    Given I am on homepage
    When I go to "/nonexistent-page"
    Then I should see "Page Not Found"
    # Navigation should still work
    When I click "Go to Homepage"
    Then I should see "Welcome"
```

How It Works:

`page.goto()` does not fail on a 404 response, the navigation succeeds, the error page renders, and your content assertions take over from there. That is exactly what you want: you are testing the error **page**, not the error **code**. The second scenario is the one teams skip and regret, it proves the user can escape. A 404 with a dead header menu is a trap, and a trap with your logo on it.

The `@regression` tag marks these scenarios for the full nightly run. Error pages change rarely, so they do not need to run on every pull request, but they absolutely need to run before a release.

At scale: A two-scenario 404 check is one of the highest-value, lowest-cost smoke tests you can standardize across a portfolio. When the same error-page feature file runs on twenty sites, a routing misconfiguration in a deploy, the kind where suddenly *everything* returns the 404 template, or the 404 returns a 500, shows up as twenty failing builds within the hour instead of a support ticket the next morning. Standardize the error-page copy across projects and the feature file needs zero per-project edits.

Recipe 39 Query Parameter Navigation

Problem: Search, filters, sorting, pagination, half of your application's interesting states live in the query string. Clicking through the UI to reach "electronics, sorted by price, page 2" takes five steps and five chances to fail. But the URL *is* the state: navigate straight to it, and the scenario tests what the filtered page shows instead of how many clicks it took to get there.

Solution:

```
Feature: Query Parameter Navigation
  Scenario: Search with query parameter
    Given I am on homepage
    When I go to "/search?q=laptop"
    And wait for AJAX to finish
    Then I should see "Search results for: laptop"

  Scenario: Filtered product listing
    Given I am on homepage
    When I go to "/products?category=electronics&sort=price"
    And wait for AJAX to finish
    Then I should see "Electronics"

  Scenario: Pagination with query parameter
    Given I am on homepage
    When I go to "/blog?page=2"
    And wait for AJAX to finish
    Then I should see "Page 2"
```

How It Works:

The query string is just part of the path argument, webship-js concatenates `launchUrl + "/search?q=laptop"` and navigates. Nothing special happens to the parameters; they reach the server exactly as written. Two things to keep in mind. First, if a parameter value contains spaces or special characters, encode it yourself (`q=gaming%20laptop`), the step passes the string through verbatim. Second, deep-linking into a parameterized state is also a real user journey: people bookmark filtered views and share search result links. If `/products?category=electronics&sort=price` renders an unfiltered page, that is a bug your click-through tests would never see.

Recipe 40 Hash/Anchor Navigation

Problem: Your FAQ page has a table of contents, your docs have "jump to section" links, and your marketing emails deep-link to `#pricing`. Anchors break quietly, a heading gets renamed, its `id` changes, and every link pointing at it now scrolls nowhere. The page still loads, so no monitoring catches it. Only a test that follows the anchor and checks what is on screen will.

Solution:

```
Feature: Anchor Navigation
  Scenario: Navigate to page section
    Given I am on homepage
    When I go to "/faq#pricing"
    Then I should see "Pricing FAQ"

  Scenario: In-page anchor links
    Given I am on homepage
    When I go to "/documentation"
    And I click "Installation"
    Then I should see "Installation Guide"

    When I click "Configuration"
    Then I should see "Configuration Options"
```

How It Works:

A `#fragment` in the URL never reaches the server, the browser loads `/faq`, then scrolls to the element whose `id` is `pricing`. There is no second page load and usually no network traffic, which is why webship-js's DOM-quiet check inside `smartSettle` matters here: it watches `window.__webshipLastMutation` rather than network activity, so lazy-rendered sections that mount as you scroll into them still get waited for. Note what the assertion really verifies: that the target content exists and is visible on the page. If you need to prove the page actually *scrolled* there, assert on something only visible in that section's expanded or highlighted state, or check the fragment with `Then the url should match "#pricing"`.

Recipe 41 Clicking Links

Problem: A new hire writes their first scenario, uses `When I click "Submit"` on a button, and watches it fail with "element not found" while the button sits right there on the screen. The click/press distinction trips up everyone exactly once. Meanwhile, the seasoned mistake is different: clicking by CSS class, which works until the next theme update renames everything. You want link clicks that read like English and survive a redesign.

Solution:

```
Feature: Link Interactions
  Scenario: Click text links
    Given I am on homepage
    When I click "About Us"
    Then I should see "Our Story"

  Scenario: Click navigation menu links
    Given I am on homepage
    When I click "Products"
    Then I should see "Our Products"

    When I click "Blog"
    Then I should see "Latest Posts"

    When I click "Contact"
    Then I should see "Get In Touch"

  Scenario: Click a link and verify destination
    Given I am on homepage
    When I click "Read More"
    Then I should see "Full Article"
    And I should not see "Read More"
```

How It Works:

`When I click "About Us"` finds an `<a>` element with the text "About Us" and clicks it. The selector strategy:

```
Priority order for finding the element:
1. Link text content: <a>About Us</a>
2. Link title attribute: <a title="About Us">...</a>
3. Link aria-label: <a aria-label="About Us">...</a>
```

Important distinction:

- `When I click "Text"`, for links (`<a>` elements)
- `When I press "Text"`, for buttons (`<button>`, `<input type="submit">`)

webship-js 2.0 also gives you role-based clicks built on Playwright's `getByRole`:

```
When I click the "About Us" link
When I click the "Save" button
When I click the "Settings" tab
```

These resolve through the accessibility tree, the same way a screen reader finds elements, so they work regardless of whether the "link" is an `<a>`, a `<button>` styled as a link, or a `<div role="link">` someone should not have written. They also double as a quiet accessibility check: if `getByRole` cannot find your link, assistive technology cannot either.

At scale: Pick one clicking convention and write it down. On a team maintaining dozens of suites, half the review friction comes from three engineers using three styles for the same action, `I click "X"`, `I click the "X" link`, and a raw CSS selector. The role-based form is the most resilient across redesigns and across projects with different markup habits, which makes it the best default for a shared convention doc.

Recipe 42 Testing Breadcrumb Navigation

Problem: A visitor lands three levels deep from a search engine, straight onto `/products/electronics/laptops`, and the breadcrumb trail is their only map of where they are. Then a category gets renamed, the trail still shows the old parent, and clicking it goes to a 404. Breadcrumbs are generated code on most platforms, which means one template bug breaks them on every page at once. Cheap to test, embarrassing to ship broken.

Solution:

```
Feature: Breadcrumb Navigation
  As a user browsing the site
  I want breadcrumbs showing my location
  So that I can navigate back to parent pages

  Scenario: Breadcrumb shows correct hierarchy
    Given I am on homepage
    When I go to "/products/electronics/laptops"
    And wait for AJAX to finish
    Then I should see "Home"
    And I should see "Products"
    And I should see "Electronics"
    And I should see "Laptops"

  Scenario: Clicking breadcrumb navigates to parent
    Given I am on homepage
    When I go to "/products/electronics/laptops"
    And wait for AJAX to finish
    And I click "Electronics"
    And wait for AJAX to finish
    Then I should see "Electronics"
    And I should see "Laptops"
    And I should see "Smartphones"
```

How It Works:

The first scenario deep-links into the hierarchy and asserts every level of the trail is present, if the breadcrumb template drops a level or renders the wrong parent, this fails immediately. The second scenario does what users actually do: click a middle crumb and verify you arrive at the parent listing (seeing both “Laptops” and “Smartphones” proves you are on the Electronics category page, not still on the laptops page). Because breadcrumbs are template-generated, two scenarios on one deep path effectively test the breadcrumb logic for the whole site.

Recipe 43 Testing Pagination Navigation

Problem: Page one of the blog always works, it is the page everyone looks at. The bugs hide at the edges: page 2 shows the same posts as page 1, the “Next” button appears on the last page and leads to an empty list, or a query change quietly breaks the offset math. Nobody clicks to page 10 manually. Your test suite should be the one user who always does.

Solution:

```
Feature: Pagination
  As a user browsing a long list
  I want to navigate between pages
  So that I can see all the items

  Background:
    Given I am on homepage
    When I go to "/blog"
    And wait for AJAX to finish

  Scenario: First page shows initial items
    Then I should see "Post 1"
    And I should see "Next"

  Scenario: Navigate to next page
    When I click "Next"
    And wait for AJAX to finish
    Then I should see "Previous"
    And I should see "Next"

  Scenario: Navigate to specific page number
    When I click "3"
    And wait for AJAX to finish
    Then I should see "Page 3"

  Scenario: Last page has no next button
    When I go to "/blog?page=10"
    And wait for AJAX to finish
    Then I should see "Previous"
    And I should not see "Next"
```

How It Works:

The `Background` runs before **each** scenario, so every scenario starts fresh on `/blog` page one, no scenario depends on where the previous one left off, which keeps each one independently re-runnable. The four scenarios cover the pagination contract: first page renders, Next advances, direct page numbers work, and the last page has no Next. That last one is the assertion teams forget, and it is the one that produces the "empty page 11" support ticket.

Notice the mix of click navigation (`When I click "Next"`) and direct URL navigation (`When I go to "/blog?page=10"`). Clicking tests the control; the URL jump tests the boundary without clicking Next nine times.

At scale: The flaky part of pagination tests is never the pagination, it is the data assumption. "Page 10 is the last page" is true on the project where you wrote the test and false on the other nineteen. Either seed a known content count as part of test setup, or keep boundary assertions structural ("Previous visible, Next absent") rather than tied to a specific page number. Suites that encode content counts from one environment are the ones that go red the first week you reuse them.

Recipe 44 Testing Sidebar/Menu Navigation

Problem: The admin panel sidebar is the workhorse navigation of your application, editors live in it eight hours a day. It is also where permission bugs surface: a menu item that 403s for the role that needs it, a section that vanished after a module update. The Tuesday before release, "can an admin still reach Settings?" should be a green checkmark in the report, not a question someone answers by logging in and looking.

Solution:

```
Feature: Sidebar Navigation
  As a user of the admin panel
  I want to navigate using the sidebar menu
  So that I can access different admin sections

  Background:
    Given I am a logged in user with the "admin" user
    And I go to "/admin"
    And wait for AJAX to finish

  Scenario: Navigate through admin sidebar
    Then I should see "Dashboard"

    When I click "Users"
    And wait for AJAX to finish
    Then I should see "User Management"

    When I click "Content"
    And wait for AJAX to finish
    Then I should see "Content Management"

    When I click "Settings"
    And wait for AJAX to finish
    Then I should see "System Settings"

  Scenario: Active menu item is highlighted
    When I click "Users"
    And wait for AJAX to finish
    Then I should see "User Management"

    When I click "Dashboard"
    And wait for AJAX to finish
    Then I should see "Dashboard"
```

How It Works:

The `Background` logs in and lands on `/admin` before each scenario, so the menu walk always starts from a known place. Each click asserts on content unique to the destination section, “User Management”, not just “Users”, which also appears in the sidebar itself. That distinction matters: asserting on text that exists in the menu you just clicked is the classic false-positive in navigation tests.

At scale: Logging in fresh in every scenario’s Background is the single biggest runtime cost in admin-panel suites, multiply a five-second login by hundreds of scenarios by forty projects and you are paying for hours of CI time that test nothing. webship-js ships auth-state steps that save and restore Playwright’s `storageState`: log in once, save the session, and restore it in every scenario after that. Chapter 10 covers the pattern in full; for a sidebar suite like this one it routinely cuts total runtime by half or more.

Navigation Recipes Quick Reference

WHAT YOU WANT	STEP TO USE
Go to homepage	Given I am on homepage
Go to a page	When I go to "/path"
Go to full URL	Given I am on "https://..."
Browser back	When I move backward one page
Browser forward	When I move forward one page
Refresh page	When I reload the page
Verify current URL	Then I should be on "/path"
Verify URL by regex	Then the url should match
	"regex"
Click a link	When I click "Link Text"
Click a button	When I press "Button Text"
Verify content visible	Then I should see "text"
Verify content NOT visible	Then I should not see "text"
Verify link URL	Then the "X" link should
	contain "/url"

Chapter Summary

- Navigation is the foundation of every web test
- `Given I am on homepage` → navigates to `launchUrl`
- `When I go to "/path"` → navigates to `launchUrl + /path`
- Use full URLs for external navigation
- Test browser history (back/forward) especially in SPAs
- Test redirects by checking page content first; pin the address with `Then I should be on "/path"` when the URL itself is the behaviour
- Test error pages (404s) to ensure good user experience
- Use `wait for AJAX to finish` after dynamic page transitions
- `click` is for links, `press` is for buttons

Next chapter: Form Testing, the most recipe-rich chapter in the book.

Chapter 6: Form Testing Recipes

Forms are the primary way users interact with web applications. Login forms, registration forms, search boxes, checkout flows, settings panels, content editors, they are everywhere. If your forms do not work, your application does not work.

This chapter is the most recipe-dense in the book. We cover every form element type, every interaction pattern, and every validation scenario you will encounter.

6.1 Text Input Fields

Recipe 45 Filling Text Fields

Problem: Registration forms are where most suites take their first real damage. The frontend team renames `id="fname"` to `id="first-name"` during a refactor, and every test that targeted the field by CSS selector dies, even though the form works perfectly. You want to fill fields the way a user finds them: by the label they can read. Then markup refactors stop being your problem.

Solution:

```
Feature: Text Input
  Scenario: Fill various text fields
    Given I am on homepage
    When I go to "/register"

    # Standard text input
    And I fill in "First Name" with "Rajab"

    # Email input
    And I fill in "Email" with "rajab@example.com"

    # Password input
    And I fill in "Password" with "SecureP@ss123"

    # Phone input
    And I fill in "Phone" with "+962-79-1234567"

    # URL input
    And I fill in "Website" with "https://natshah.com"

    And I press "Register"
    And wait for AJAX to finish
    Then I should see "Registration successful"
```

How It Works:

`When I fill in "First Name" with "Rajab"` resolves the field **label-first**, then falls back to id, name, and placeholder:

```
Field Resolution (label-first, tried in order):

1. Label text:                <label for="fname">First Name</label>
                              <input id="fname" />

2. ID attribute:              <input id="first-name" />

3. Name attribute: <input name="first_name" />

4. Placeholder:               <input placeholder="First Name" />

                      The FIRST match wins. Label-first means your step targets
what the USER sees, ids and names can change in a refactor,
but the visible label rarely does.
```

FIGURE 6.1 *Filling Text Fields (Recipe 45)*

Label-first resolution is a deliberate design choice. The label is part of the user-facing contract of the form; the `id` is an implementation detail. When a test fills in “First Name”, it documents the form the way a product owner would describe it, and it survives the frontend refactor that renames every attribute underneath.

Two variants worth knowing. For long forms, the table form fills everything in one step:

```
When I fill in the following:
  | First Name | Rajab              |
  | Email      | rajab@example.com  |
  | Phone      | +962-79-1234567    |
```

And when a field genuinely has no usable label, custom widgets, generated markup, you can target a test attribute directly: `When I fill in "user-email" with "rajab@example.com" by its "data-test" attribute`.

At scale: Across many projects, the rule that pays for itself is: fill by label, and if there is no label, that is a bug to file, not a CSS selector to write. Unlabeled inputs fail accessibility audits too (Chapter 15), so the test convention and the a11y requirement reinforce each other. Teams that adopt `data-test` attributes as the agreed fallback get a second win: frontend developers know exactly which attributes the test suite depends on, and code review can protect them.

Recipe 46 Verifying Input Values

Problem: The edit form loads, the success toast appeared yesterday, everyone moved on, and the Username field is quietly empty because the API response changed shape and the form no longer pre-fills. “I should see” cannot catch this: input values live in the `value` property, not in the page text, so a text assertion looks straight past an empty field. You need an assertion that reads what is actually inside the input.

Solution:

```
Feature: Input Value Verification
  Scenario: Verify pre-filled form data
    Given I am a logged in user with the "admin" user
    When I go to "/profile/edit"
    And wait for AJAX to finish
    Then I should see "admin" value in the "Username" input element

  Scenario: Verify saved form data
    Given I am a logged in user with the "admin" user
    When I go to "/settings"
    And I fill in "Display Name" with "Tester Demo"
    And I press "Save"
    And wait for AJAX to finish
    When I reload the page
    And wait for AJAX to finish
    Then I should see "Tester Demo" value in the "Display Name" input
element
```

How It Works:

This step resolves the field the same label-first way as `I fill in`, then reads its `value` property, the live state of the input, not the HTML attribute it started with. The second scenario is the pattern to memorize: **fill, save, reload, verify the value**. Without the reload you are only testing that typing works; with it you are testing that the save round-tripped through the server. There is also a selector-based, web-first variant, `Then "#display-name" should have value "Tester Demo"`, which polls every 100ms for up to 5 seconds, useful when the value arrives asynchronously after the page settles.

Recipe 47 Testing Placeholder Text

Problem: The design team rewrites the search placeholder from “Search products…” to something less helpful, or a translation update wipes it entirely, and nobody notices because placeholders are not content anyone reviews. Here is the trap when you go to test it: a placeholder is **not page text**. It is an attribute the browser paints into an empty field, so `Then I should see` looks straight past it. You need an assertion that reads the attribute itself.

Solution:

```
Feature: Placeholder Text
  Scenario: Search field has helpful placeholder
    Given I am on homepage
    Then "#search" should have attribute "placeholder" with value "Search
products..."
```

How It Works:

`should have attribute "placeholder" with value "..."` is one of webship-js’s web-first assertions: it polls the element every 100ms for up to 5 seconds, so a search box that mounts late (a JavaScript-rendered header, for instance) still passes without any explicit wait. Asserting on the attribute rather than on rendered text means the test passes whether the field is empty or focused, a text-based check would flicker depending on field state.

Pro Tip: Placeholder checks earn their keep on multilingual sites. The placeholder is one of the easiest strings to miss in a translation pass, and one attribute assertion per locale catches it.

Recipe 48 Testing Character Limits

Problem: The database column says `VARCHAR(30)`, the form says nothing, and the first user with a long name gets a raw SQL error on a Friday afternoon. Character limits live in three places, the HTML `maxlength`, the client-side validator, and the server, and they drift apart over time. The only way to know which one actually fires is to push an oversized value through the form and see what comes back.

Solution:

```
Feature: Character Limits
  Scenario: Username has maximum length
    Given I am on homepage
    When I go to "/register"
    And I fill in "Username" with
"ThisIsAVeryLongUsernameExceedingTheMaximumAllowedCharacterLimitForThisFie
ld"
    And I press "Register"
    Then I should see "Username must be 30 characters or less"

  Scenario: Bio field allows longer text
    Given I am a logged in user with the "admin" user
    When I go to "/profile/edit"
    And I fill in "Bio" with "I am a software engineer with 15 years of
experience in web development, automated testing, and DevOps. I specialize
in BDD and continuous integration."
    And I press "Save"
    And wait for AJAX to finish
    Then I should see "Profile updated"
```

How It Works:

One subtlety: if the input has an HTML `maxlength="30"`, the browser silently truncates as the step types, the server never sees the long value, and the error message scenario will fail because no error appears. That is not a broken test; it is information. It tells you the client enforces the limit. To verify the truncation itself, follow up with `Then "#username" should have value` and the 30-character prefix. The second scenario is the often-forgotten mirror image: fields that **should** accept long text. A too-aggressive validation rule that rejects a normal-length bio is just as much a bug as a missing one.

6.2 Select Dropdowns

Recipe 49 Selecting Options by Text

Problem: A backend developer reorders the country list, or swaps the option values from ISO codes to database IDs, and every test that selected options by value or by index starts choosing the wrong country, silently. The user picks “Canada” by reading the word “Canada”, and your test should too. Select by visible text, and value refactors stop reaching your suite.

Solution:

```
Feature: Dropdown Selection
  Scenario: Select country from dropdown
    Given I am on homepage
    When I go to "/profile/edit"
    And I select "Canada" from "Country"
    And I press "Save"
    And wait for AJAX to finish
    Then I should see "Profile updated"

  Scenario: Select multiple options in sequence
    Given I am on homepage
    When I go to "/register"
    And I select "Mr." from "Title"
    And I select "Canada" from "Country"
    And I select "English" from "Preferred Language"
    And I select "UTC+3" from "Timezone"
    And I press "Register"
    And wait for AJAX to finish
    Then I should see "Registration successful"
```

How It Works:

`When I select "Canada" from "Country"` finds a `<select>` element labeled “Country”, the same label-first resolution as text fields, and selects the `<option>` with visible text “Canada”.

```
<!-- This is what the step matches: -->
<label for="country">Country</label>
<select id="country" name="country">
   <option value="">-- Select --</option>
   <option value="CA">Canada</option>                <!--   ◀ selected by "Canada" -->
   <option value="US">United States</option>
   <option value="UK">United Kingdom</option>
</select>
```

FIGURE 6.2 *Selecting Options by Text (Recipe 49)*

Selecting fires the `change` event, exactly as a real selection would, so any JavaScript listening for it (cascading dropdowns, conditional fields) runs as normal.

For multi-selects there is a companion step: `When I additionally select "Arabic" from "Languages"` adds to the current selection instead of replacing it. The plain `I select` step clears previous choices first; `I additionally select` does not. Mixing them up is the classic reason a multi-select test ends with only the last option selected.

At scale: Option **labels** are also translations. On multilingual projects the visible text changes per locale while the values stay fixed, which flips the advice: a suite that must run across language variants should pin the interface language first (or use a locale-specific examples table), or your "Germany" becomes "Deutschland" and the step finds nothing. Decide per portfolio: one canonical test locale for shared suites, with a thin per-locale layer on top for translation checks.

Recipe 50 Testing Dynamic/AJAX Dropdowns

Problem: Country/city cascades are the birthplace of flaky form tests. Select "Canada", and the city dropdown takes 300ms to fetch its options, on your laptop. In CI, under load, it takes 2 seconds, and the test that "always passed locally" picks from an empty dropdown and fails. The fix is never a sleep. It is waiting for the request that fills the dropdown to actually finish.

Solution:

```
Feature: Dynamic Dropdowns
  Scenario: Country dropdown loads states/provinces
    Given I am on homepage
    When I go to "/address"
    And I select "Canada" from "Country"
    And wait for AJAX to finish
    And I select "Toronto" from "City"
    And I press "Save"
    And wait for AJAX to finish
    Then I should see "Address saved"
```

How It Works:

The key is `And wait for AJAX to finish` between selecting the country and selecting the city. When "Canada" is selected, an AJAX request fetches Canada's cities. Without the wait, the city dropdown would still be empty when the test tries to select "Toronto."

FIGURE 6.3 *Testing Dynamic/AJAX Dropdowns (Recipe 50)*

Under the hood this is webship-js's instrumentation at work. Every page gets a fetch/XHR wrapper installed via `addInitScript` that maintains `window.__webshipAjaxCount`, incremented when a request starts, decremented when it completes. The wait step polls every 50ms until that counter is zero **and** no wrapped timers are pending **and** the DOM has been quiet for at least 250ms (the `__webshipLastMutation` timestamp). So it does not just wait for the network response, it waits for the JavaScript that *renders the options into the dropdown* to finish too. That last part is what naive "wait for response" approaches miss.

In practice you often get away without the explicit step, because webship-js runs `smartSettle(page, 1500)` after every state-changing step anyway. Keep the explicit wait for cascades regardless: a slow staging API can outlast the 1500ms AfterStep budget, and the explicit step gets the full window.

At scale: Cascading-dropdown timing is environment-dependent, which makes it the number-one source of "passes on project A, flakes on project B" reports when a shared suite rolls out. Because webship-js waits on the AJAX counter rather than a duration, the same feature file absorbs a 100ms dev API and a 3-second staging API without edits, no per-project timeout tuning, no `sleep 2` creeping into the repo. If a cascade still flakes, the finding is real: the app is rendering options before its own request completes.

6.3 Checkboxes and Radio Buttons

Recipe 51 Checking and Unchecking Checkboxes

Problem: Notification preferences, consent boxes, feature toggles, checkboxes carry more legal and business weight than their size suggests. The bug pattern is sneaky: a test that *clicks* a checkbox toggles it, so if the default state ever changes, the test starts silently unchecking the thing it meant to check. You want steps that declare the end state, not the gesture.

Solution:

```
Feature: Checkbox Interactions
  Scenario: Check multiple checkboxes
    Given I am on homepage
    When I go to "/preferences"
    And I check "Email notifications"
    And I check "SMS alerts"
    And I check "Newsletter subscription"
    And I press "Save Preferences"
    And wait for AJAX to finish
    Then I should see "Preferences saved"

  Scenario: Uncheck a checkbox
    Given I am a logged in user with the "admin" user
    When I go to "/settings"
    And I uncheck "Show email on profile"
    And I press "Save"
    And wait for AJAX to finish
    Then I should see "Settings saved"
```

How It Works:

`When I check "Email notifications"` resolves the checkbox by its label (the same label-first strategy as text fields) and ensures it ends up **checked**, if it is already checked, nothing happens. `I uncheck` is the mirror image. This idempotence is the whole point: the steps state an outcome, not a click. A toggle-by-click test depends on the starting state; a check/uncheck test does not, which is exactly what you want when defaults differ between fresh installs and seeded test environments. Both steps fire the `change` event, so dependent UI (a "notification frequency" field that appears when notifications are on) reacts normally.

Recipe 52 Verifying Checkbox State

Problem: A privacy-sensitive default flips during a dependency update, "Show email on profile" ships checked when it must ship unchecked. Nobody changed that code on purpose; a form library upgrade changed how defaults are applied. Default checkbox states are invisible in code review and devastating in production, which makes them perfect candidates for an explicit assertion.

Solution:

```
Feature: Checkbox State Verification
  Scenario: Verify default checkbox states
    Given I am a logged in user with the "admin" user
    When I go to "/settings"
    Then the "Email notifications" checkbox is checked
    And the "SMS alerts" checkbox is checked

  Scenario: Verify checkbox state after change
    Given I am a logged in user with the "admin" user
    When I go to "/settings"
    And I uncheck "Email notifications"
    And I press "Save"
    And wait for AJAX to finish
    When I reload the page
    And wait for AJAX to finish
    # Checkbox should now be unchecked (persisted)
    Then I should see "Settings"
```

How It Works:

`Then the "Email notifications" checkbox is checked` reads the live `checked` property of the input, not the HTML attribute, which only reflects the initial state. The second scenario shows the persistence pattern again (change, save, reload), and the comment marks where you would strengthen it: after the reload, assert the actual state with the checkbox step rather than settling for "the page loaded". A persistence test that only checks the page title is a navigation test wearing a costume.

At scale: Defaults are where a product owner's acceptance criteria and your suite meet most directly. "New users get email notifications on, SMS off" is a one-line business rule that fits in two assertion steps, readable in the HTML report by the same PO who wrote the criterion. Across a portfolio, a short `defaults.feature` per project turns "what are the defaults supposed to be?" from an archaeology exercise into a green checkmark.

Recipe 53 Selecting Radio Buttons

Problem: Shipping method, payment type, account tier, radio groups drive price and behaviour downstream, and a misfired selection corrupts everything after it in the checkout. Radios are also fiddly to automate by hand: the visible "button" is often a styled label while the real input is hidden off-screen, so naive element clicks miss. You want a step that understands radio semantics.

Solution:

```
Feature: Radio Button Selection
  Scenario: Select shipping method
    Given I am on homepage
    When I go to "/checkout/shipping"
    And I select radio button "Express Shipping"
    Then the "Express Shipping" radio button is selected

  Scenario: Change radio button selection
    Given I am on homepage
    When I go to "/checkout/shipping"
    And I select radio button "Standard Shipping"
    Then the "Standard Shipping" radio button is selected

    When I select radio button "Express Shipping"
    Then the "Express Shipping" radio button is selected
    # Standard is no longer selected (only one can be selected)
```

How It Works:

`When I select radio button "Express Shipping"` finds the radio by its label text and selects it, and because labels are clickable proxies for their inputs, this works even when the actual `<input type="radio">` is visually hidden behind a styled control, which is how most design systems build radios today. The second scenario tests the part of radio semantics that custom-built widgets get wrong most often: selecting one option must deselect the others. With native radios the browser guarantees it; with a JavaScript "radio group" made of divs, only your test does.

Recipe 54 Verifying Radio Button State

Problem: The checkout defaults to Standard Shipping, until a template change defaults it to Overnight and customers start paying triple without noticing the pre-selection. Default radio state is a business decision encoded in one HTML attribute, and it deserves a test with its name on it.

Solution:

```
Feature: Radio Button State
  Scenario: Verify default selection
    Given I am on homepage
    When I go to "/checkout/shipping"
    Then the "Standard Shipping" radio button is selected

  Scenario Outline: Verify radio options
    Given I am on homepage
    When I go to "/checkout/shipping"
    And I select radio button "<option>"
    Then the "<option>" radio button is selected

    Examples:
      | option             |
      | Standard Shipping  |
      | Express Shipping   |
      | Overnight Shipping |
```

How It Works:

The first scenario pins the default, no interaction, just “load the page and check which radio the user starts on”. The Scenario Outline then runs once per Examples row, each as an independent scenario with a fresh page, proving every option in the group is actually selectable. That matters more than it sounds: a typo in one option’s `name` attribute quietly splits it out of the radio group, and it becomes the option that can be clicked but never stays selected. Three rows in a table is cheaper than the support ticket.

6.4 Rich Text Editors

Recipe 55 Filling CKEditor / Rich Text Fields

Problem: Your first attempt at testing a CKEditor field is always the same story: `I fill in "Body"` cannot find the field, because the `<textarea>` it targets is hidden, the editor replaced it with its own editing surface. Then someone tries keyboard-typing into the surface, and it works until the editor's autocomplete, paste filtering, or toolbar state makes it flaky. Content teams live in these editors all day; your suite needs a way in that does not fight the editor.

Solution:

```
Feature: Rich Text Editor
  Scenario: Create a blog post with formatted content
    Given I am a logged in user with the "editor" user
    When I go to "/admin/posts/new"
    And wait for AJAX to finish
    And I fill in "Title" with "My BDD Journey"
    And I fill the rich text editor field "Body" with "<p>Today I started
learning <strong>BDD</strong> with webship-js.</p>"
    And I press "Publish"
    And wait for AJAX to finish
    Then I should see "Post published"

  Scenario: Rich text with complex HTML
    Given I am a logged in user with the "editor" user
    When I go to "/admin/posts/new"
    And wait for AJAX to finish
    And I fill in "Title" with "Feature List"
    And I fill the rich text editor field "Body" with "<h2>Features</h2>
<ul><li>Easy to use</li><li>Powerful automation</li><li>Beautiful
reports</li></ul><p>Try it <em>today</em>!</p>"
    And I press "Publish"
    And wait for AJAX to finish
    Then I should see "Post published"
```

How It Works:

Rich text editors (CKEditor, TinyMCE, Quill, Froala, etc.) replace standard `<textarea>` elements with complex JavaScript-powered editing surfaces. They use either:

- `contenteditable` divs
- iframes with their own document
- Shadow DOM elements

The `I fill the rich text editor field` step handles this complexity by:

1. Finding the editor instance by its field name
2. Accessing the editor's internal API
3. Setting the content programmatically

This is more reliable than trying to type into the editor, because keyboard input in editors can behave unpredictably in automated tests, autocorrect, paste filters, and toolbar state all interfere with simulated keystrokes, and they interfere differently across editor versions.

Setting content through the editor's own API also means the editor's change events fire, so "unsaved changes" warnings, autosave timers, and character counters all behave as if a human had typed. Those autosave timers, by the way, are exactly the kind of `setTimeout` work that `window.__webshipPendingTimers` tracks, which is why webship-js's settle logic does not move on while the editor is still mid-autosave.

At scale: Rich text editors are where CMS portfolios feel version drift first. We learned this maintaining tests across dozens of Varbase sites: a CKEditor major upgrade changes the DOM completely, and every suite that poked at editor internals with CSS selectors broke at once, while suites using the field-name step kept passing, because the step targets the field, not the editor's markup. Keep editor interaction behind this one step, and an editor upgrade becomes one framework update instead of fifty broken feature files.

Recipe 56 Mixing Rich Text with Regular Fields

Problem: Real content forms are a zoo: a title field, a slug, two rich text editors, a category drop-down, a featured checkbox, and an image upload, all on one screen, all required before “Publish” does anything. This is the form your editors fill thirty times a week, which makes it the single most valuable form to automate, and the one where each element type needs its matching step.

Solution:

```
Feature: Complete Content Form
  Scenario: Create a full article
    Given I am a logged in user with the "editor" user
    When I go to "/admin/articles/new"
    And wait for AJAX to finish

    # Regular text fields
    And I fill in "Title" with "Getting Started with Automated Testing"
    And I fill in "Slug" with "getting-started-automated-testing"
    And I fill in "Meta Description" with "Learn how to set up automated
tests"

    # Rich text editors
    And I fill the rich text editor field "Summary" with "<p>A beginner-
friendly guide to automated testing.</p>"
    And I fill the rich text editor field "Body" with "
<h2>Introduction</h2><p>Automated testing saves time and catches bugs
early.</p><h2>Getting Started</h2><p>Install webship-js and write your
first test in minutes.</p>"

    # Dropdown
    And I select "Technology" from "Category"

    # Checkbox
    And I check "Featured Article"

    # File upload
    And I attach the file "test-image.png" to "Featured Image"

    # Submit
    And I press "Publish"
    And wait for AJAX to finish
    Then I should see "Article published successfully"
```

How It Works:

Each element type gets its specialist step, but they all share the same label-first resolution, so the scenario reads like the form itself, top to bottom. The comments grouping the steps are not decoration: when this scenario fails at step 9 of 12, the report shows exactly which group broke, and a content editor can read the scenario and confirm it matches their real workflow. That review, "yes, this is how we publish", is worth more than any code review for catching a missing required field. For the three plain text fields at the top, the table step (`When I fill in the following:`) would compress them to one step; with a mixed form like this, many teams prefer the explicit one-step-per-field version because the diff stays readable when the form changes.

6.5 File Uploads

Recipe 57 Single File Upload

Problem: File uploads stop manual-testing habits dead: the OS file picker is a native dialog the browser cannot script, so "just click Browse and pick a file" does not translate to automation. Meanwhile the upload path, validation, storage, thumbnail generation, is some of the most failure-prone code in any application. You need a way to hand the browser a file without ever opening the dialog.

Solution:

```
Feature: File Upload
  Scenario: Upload a profile picture
    Given I am a logged in user with the "admin" user
    When I go to "/profile/edit"
    And I attach the file "test-image.png" to "profile_picture"
    And I press "Upload"
    And wait for AJAX to finish
    Then I should see "Profile picture updated"
```

How It Works:

`When I attach the file "test-image.png" to "profile_picture"` does the following:

1. Looks for the file at `{assets_folder}/test-image.png`
2. Finds the file input element (`<input type="file" name="profile_picture">`)
3. Uses Playwright's `setInputFiles()` to attach the file

`setInputFiles()` is the trick that makes this possible: it sets the input's file list directly at the protocol level and fires the `change` event, so the page reacts exactly as if a user had picked the file, preview thumbnails render, upload progress starts, but no native OS dialog ever opens. That dialog is the reason generic click automation cannot do uploads; Playwright simply routes around it.

The file must exist in your `assets_folder` (configured in `cucumber.js`). If the file is not found, the step fails with a clear error message, before touching the browser, which makes a missing asset easy to tell apart from a broken upload.

At scale: Keep the test assets folder small, versioned, and shared. A handful of canonical files, a tiny valid PNG, an oversized image, a PDF, a file with a unicode name, covers nearly every upload scenario on every project, and committing them to the suite repository means CI runners and new laptops get identical bytes. Resist real-world-sized files: a 4MB "realistic" image uploaded by forty nightly suites is pure CI cost with zero added coverage. Test size limits with a file just over the limit, not a huge one.

Recipe 58 Testing Upload Validation

Problem: Someone uploads a PDF where the avatar goes, and either the app politely refuses, or the image pipeline chokes, the profile page renders a broken image for every visitor, and you find out from a screenshot in Slack. Upload validation is written once and then forgotten; meanwhile file-type checks are also a security boundary. The rejection paths deserve the same coverage as the happy one.

Solution:

```
Feature: Upload Validation
  Background:
    Given I am a logged in user with the "admin" user
    When I go to "/profile/edit"

  Scenario: Upload valid image
    When I attach the file "test-image.png" to "profile_picture"
    And I press "Upload"
    And wait for AJAX to finish
    Then I should see "Profile picture updated"

  Scenario: Upload invalid file type
    When I attach the file "sample.pdf" to "profile_picture"
    And I press "Upload"
    Then I should see "Only image files are allowed"

  Scenario: Upload with no file selected
    When I press "Upload"
    Then I should see "Please select a file"
```

How It Works:

Three scenarios, three different code paths: acceptance, type rejection, and the empty submit. Notice the rejection scenarios skip `wait for AJAX to finish`, client-side validation answers instantly, and if your app validates file type on the server instead, webship-js's automatic post-step settle covers the round trip anyway. The `Background` keeps each scenario independent: each one starts freshly logged in on the edit page, so the PDF rejection cannot leak state into the valid-upload run. One more case worth adding when your app has size limits: a file just over the limit, expecting the size message, it completes the validation triangle of type, size, and presence.

6.6 Form Submission and Validation

Recipe 59 Submit and Verify Success

Problem: The contact form is often a company's single biggest lead source, and it breaks in the most boring ways, a misconfigured mail service, a CSRF token expiry, a JavaScript error swallowing the submit. Each time, it breaks silently: the page looks fine, the user clicks Send, and nothing arrives. The Tuesday before a release, "does the contact form still work, end to end?" should be answered by a tagged scenario, not by someone sending themselves a test message.

Solution:

```
Feature: Contact Form
  @smoke
  Scenario: Submit a complete contact form
    Given I am on homepage
    When I go to "/contact"

    # Fill all fields
    And I fill in "Name" with "Tester Demo"
    And I fill in "Email" with "rajab@example.com"
    And I select "Technical Support" from "Department"
    And I fill in "Subject" with "Question about testing"
    And I fill in "Message" with "How do I test modals with webship-js?"

    # Submit
    And I press "Send Message"
    And wait for AJAX to finish

    # Verify success
    Then I should see "Thank you for your message"
    And I should see "We will respond within 24 hours"
    And I should not see "Error"
```

How It Works:

The shape is the canonical form test: fill everything, submit, then assert **three** things, the specific success message, a second confirmation detail, and the absence of an error. That last negative assertion matters more than it looks: plenty of broken forms show the thank-you banner *and* an error box, because the banner is rendered optimistically before the request resolves. Only the trio together proves a clean submission.

The `@smoke` tag marks this as a scenario that runs on every deploy, not just nightly. A form this important earns a place in the fastest feedback loop you have.

At scale: Tags are the contract between QA and the pipeline, agree on a portfolio-wide vocabulary (`@smoke` on every deploy, `@regression` nightly, `@critical` blocking release) and enforce it in review, because the day every project means the same thing by `@smoke`, your CI configuration

becomes one shared template instead of forty bespoke ones. It also gives delivery managers a sentence they can use: “all @smoke scenarios green on all projects” is a release-readiness statement that needs no translation.

Recipe 60 Testing Required Field Validation

Problem: A frontend rewrite ships, the form looks identical, and one `required` attribute fell off the Email field during the rebuild. Now the backend receives contact submissions with no reply address. Required-field rules are individually trivial and collectively fragile, every form rebuild, library swap, or field rename is a chance to drop one. The empty-submit test is thirty seconds to write and catches all of them at once.

Solution:

```
Feature: Required Field Validation
  Scenario: Submit form with missing required fields
    Given I am on homepage
    When I go to "/contact"
    And I press "Send Message"
    Then I should see "Name is required"
    And I should see "Email is required"
    And I should see "Message is required"

  Scenario: Submit form with partial data
    Given I am on homepage
    When I go to "/contact"
    And I fill in "Name" with "Rajab"
    And I press "Send Message"
    Then I should not see "Name is required"
    And I should see "Email is required"
    And I should see "Message is required"
```

How It Works:

The first scenario submits a completely empty form and expects every required-field message at once, one scenario, full coverage of the rule set. The second is the sharper instrument: fill exactly one field and assert its error *disappears* while the others *remain*. That `should not see "Name is required"` line catches a validation system that dumps every error regardless of input, and the surviving errors catch one that clears too eagerly. Together the two scenarios pin the per-field behaviour, not just "some red text appeared". No AJAX wait is needed when validation runs client-side, the messages render synchronously on submit.

Recipe 61 Testing Email Validation

Problem: Someone tightens the email regex to stop spam signups and accidentally rejects every address with a `+` tag, including the QA team's own test accounts. Email validation has two failure directions: accepting garbage and rejecting real addresses, and the second one quietly costs you users. Testing it one scenario per address would be fifty lines of copy-paste; a Scenario Outline makes it a table.

Solution:

```
Feature: Email Validation
  Scenario Outline: Email format validation
    Given I am on homepage
    When I go to "/register"
    And I fill in "Email" with "<email>"
    And I press "Submit"
    Then I should see "<message>"

    Examples: Invalid emails
      | email               | message                     |
      | plaintext           | Please enter a valid email |
      | missing@tld         | Please enter a valid email |
      | @no-local.com       | Please enter a valid email |
      | user @space.com     | Please enter a valid email |
      | .start@example.com  | Please enter a valid email |

    Examples: Valid emails
      | email                     | message              |
      | user@example.com          | Registration successful |
      | user.name@example.com     | Registration successful |
      | user+tag@example.com      | Registration successful |
      | user@subdomain.example.com | Registration successful |
```

How It Works:

Each Examples row expands into a full, independent scenario, fresh navigation, fresh form, one email, one expected message. The two named Examples blocks are the documentation trick: “Invalid emails” and “Valid emails” appear as group headings in the HTML report, so a product owner reviewing it sees the validation policy laid out as a table they could have written themselves. The valid block is the half teams skip, `user+tag@example.com` is the canary for over-tight regexes, and it belongs in every email validation table.

At scale: Remember the multiplication: nine rows × full browser scenario × every project that inherits this feature file. Keep the table to *representative* cases, one per failure class, not every malformed string you can imagine. Exhaustive format checking belongs in a unit test against the val-

idator function, where a thousand cases cost milliseconds; the browser table only needs to prove the validator is wired to the form and the message reaches the user.

Recipe 62 Testing Custom Validation Messages

Problem: “Invalid input” tells the user nothing; “Password must be at least 8 characters” tells them exactly what to fix. Your team agreed on helpful, specific messages, and then a library upgrade replaced half of them with its generic defaults, and signups dipped because nobody could tell why their password was refused. The exact wording is part of the user experience. Assert it.

Solution:

```
Feature: Custom Validation
  Scenario: Password strength requirements
    Given I am on homepage
    When I go to "/register"
    And I fill in "Password" with "weak"
    And I press "Register"
    Then I should see "Password must be at least 8 characters"

  Scenario: Password match validation
    Given I am on homepage
    When I go to "/register"
    And I fill in "Password" with "StrongPass123!"
    And I fill in "Confirm Password" with "DifferentPass123!"
    And I press "Register"
    Then I should see "Passwords do not match"
```

How It Works:

Asserting the exact message text does two jobs. It verifies the validation logic fired (the rule works), and it verifies the *copy* survived (the user gets the helpful version, not a fallback). The cross-field check in the second scenario, password versus confirmation, is worth singling out because it is the validation type most often implemented only on the client; if your server skips it, an API user can create a mismatched account. Pair this UI scenario with an API-level check (Chapter 11) when the rule is security-relevant.

6.7 Complex Form Patterns

Recipe 63 Multi-Step Wizard Forms

Problem: Wizards multiply everything that can go wrong with a form by the number of steps, then add transitions on top. A user abandons registration at step 2 of 3, and analytics cannot tell you whether the form was too long or simply broken. When a wizard breaks, it usually breaks *between* steps, the Next button saves but does not advance, or advances without saving, and only a test that walks the whole path catches it.

Solution:

```
Feature: Registration Wizard
  As a new user
  I want to complete a step-by-step registration
  So that I can set up my account properly

  Scenario: Complete multi-step registration
    Given I am on homepage
    When I go to "/register"

    # Step 1: Account Details
```

```
Then I should see "Step 1 of 3"
And I should see "Account Details"
When I fill in "Email" with "new@example.com"
And I fill in "Password" with "SecurePass123!"
And I fill in "Confirm Password" with "SecurePass123!"
And I press "Next"
And wait for AJAX to finish

# Step 2: Personal Information
Then I should see "Step 2 of 3"
And I should see "Personal Information"
When I fill in "First Name" with "Rajab"
And I fill in "Last Name" with "Natshah"
And I select "Canada" from "Country"
And I press "Next"
And wait for AJAX to finish

# Step 3: Preferences
Then I should see "Step 3 of 3"
And I should see "Preferences"
When I check "Email notifications"
And I select "English" from "Language"
And I check "I agree to the Terms of Service"
And I press "Create Account"
And wait for AJAX to finish

# Confirmation
Then I should see "Welcome, Rajab!"
And I should see "Your account has been created"
```

How It Works:

The pattern is **assert, act, assert**: confirm you are on the step you think you are on (`Then I should see "Step 1 of 3"`) *before* filling anything, then fill, advance, and confirm the next step. Without those position assertions, a wizard that silently fails to advance produces the worst kind of failure, the test types step 2's data into step 1's fields and reports an error three steps away from the actual bug. Each `wait for AJAX to finish` covers the step transition, which on most wizards is an AJAX save; webship-js's AJAX counter ensures the next assertion only runs once the new step has actually rendered.

The final assertion, "Welcome, Rajab!", closes the loop: data entered at step 2 surfaces after step 3, proving the wizard carried state across all transitions, not just displayed three screens.

At scale: One long scenario per wizard is the right call, resist splitting it into per-step scenarios, because steps 2 and 3 cannot be reached without doing step 1 anyway, and three scenarios that each replay the prefix cost triple the runtime for the same coverage. The wizard is a single user behaviour: getting registered. Tag it `@critical` and let it block releases; a broken registration wizard is a stop-everything bug on any project.

Recipe 64 Testing Wizard Back Navigation

Problem: A user gets to step 3 of registration, realizes they typoed their email, clicks Previous, and step 1 is blank. They close the tab. Losing a user's typed data on back-navigation is one of the most rage-inducing bugs a form can have, and it never shows up in forward-only testing, which is the only testing wizards usually get.

Solution:

```
Feature: Wizard Navigation
  Scenario: Go back preserves data
    Given I am on homepage
    When I go to "/register"

    # Fill Step 1
    And I fill in "Email" with "test@example.com"
    And I press "Next"
    And wait for AJAX to finish

    # Fill Step 2
    And I fill in "First Name" with "Rajab"
    And I press "Next"
    And wait for AJAX to finish

    # Go back to Step 2
    When I press "Previous"
    And wait for AJAX to finish
    Then I should see "Step 2 of 3"
    # First Name should still be filled
    Then I should see "Rajab" value in the "First Name" input element

    # Go back to Step 1
    When I press "Previous"
    And wait for AJAX to finish
    Then I should see "Step 1 of 3"
    # Email should still be filled
    Then I should see "test@example.com" value in the "Email" input
element
```

How It Works:

The crucial assertions here are the **value** checks, not the step indicators. “Step 2 of 3” appearing proves navigation works; `"Rajab" value in the "First Name" input element` proves the wizard re-hydrated the field from wherever it stores in-progress state. Those are separate mechanisms, and the second one is the one that breaks. The test walks back through *every* step it visited, because wizards commonly persist the most recent step’s data and lose the earlier ones, a partial bug that a single Previous click would miss.

Recipe 65 Conditional Form Fields

Problem: Select “Business” and three company fields are supposed to appear; select “Personal” and they must vanish, along with their `required` flags, or personal users get blocked by validation on fields they cannot see. Conditional field logic is rewritten every time the form library changes, and its failure mode is invisible: the form renders, looks complete, and is wrong. Both directions need a test, because show-but-never-hide is the most common way this logic decays.

Solution:

```
Feature: Conditional Fields
  Scenario: Company fields appear for business accounts
    Given I am on homepage
    When I go to "/register"
    And I select "Business" from "Account Type"
    And wait for AJAX to finish
    Then I should see "Company Name"
    And I should see "Tax ID"
    And I should see "Company Address"

  Scenario: Company fields hidden for personal accounts
    Given I am on homepage
    When I go to "/register"
    And I select "Personal" from "Account Type"
    And wait for AJAX to finish
    Then I should not see "Company Name"
    And I should not see "Tax ID"
```

How It Works:

The `change` event fired by `I select` triggers the show/hide logic, and the settle that follows gives the DOM time to finish mutating, webship-js's mutation observer (`window.__webshipLastMutation`) keeps the wait alive until the DOM has been quiet for 250ms, which covers both instant toggles and animated reveals. The negative assertions in the second scenario are doing the real work: "Personal account does not ask for a Tax ID" is a business rule, and `I should not see` is how a business rule about absence becomes executable. If the fields are hidden with CSS rather than removed from the DOM, the selector-level web-first form, `Then "#tax-id" should be hidden`, makes the same assertion with surgical precision.

At scale: Conditional logic is exactly the kind of acceptance criterion product owners write naturally, "if the user picks Business, ask for company details", and these two scenarios are that sentence, verbatim, in executable form. Get POs to write conditional-field rules as Given/When/Then in the ticket itself; on teams that do this, QA stops reverse-engineering intent from mockups and the scenario lands in the suite the same sprint the feature ships.

Recipe 66 Auto-Complete/Suggestion Fields

Problem: Auto-complete fields are a two-part timing puzzle: the app waits for you to stop typing (a debounce timer), then fires a search request, then renders suggestions. A test that types and immediately looks for "Laptop" loses the race on slow CI runners every time, the classic test that passes locally and flakes in the pipeline.

Solution:

```
Feature: Auto-Complete
  Scenario: Select from auto-complete suggestions
    Given I am on homepage
    When I go to "/search"
    And I fill in "Search" with "lap"
    And wait for AJAX to finish
    Then I should see "Laptop"
    And I should see "Laptop Stand"
    When I click "Laptop"
    And wait for AJAX to finish
    Then I should see "Laptop Details"
```

How It Works:

This recipe is where webship-js's timer tracking earns its keep. The debounce in an auto-complete is a `setTimeout`, and webship-js wraps `setTimeout` via `addInitScript`, counting pending timers in `window.__webshipPendingTimers`. The settle logic refuses to declare the page quiet while a timer is pending **or** an AJAX request is in flight **or** the DOM is still mutating. So the wait covers the whole chain, debounce, fetch, suggestion render, in one step, with no magic number that matches your laptop but not your CI runner. Then `When I click "Laptop"` selects the suggestion exactly as a user would: by clicking its visible text.

Pro Tip: Type a *prefix* ("lap"), not the full term. Asserting that two different suggestions appear ("Laptop" and "Laptop Stand") proves the suggestion engine is matching, not just echoing your input back.

6.8 Form Recipes Quick Reference

FORM ACTION	WEBSHIP-JS STEP
Fill text input	I fill in "Field" with "value"
Fill many fields at once	I fill in the following: (table)
Select dropdown option	I select "Option" from "Field"
Add to a multi-select	I additionally select "Option"
	from "Field"
Check a checkbox	I check "Label"
Uncheck a checkbox	I uncheck "Label"
Select radio button	I select radio button "Label"
Upload a file	I attach the file "f" to "Field"
Fill rich text editor	I fill the rich text editor
	field "Name" with "HTML"
Click a button (submit)	I press "Button Text"
Click a link	I click "Link Text"
Verify text visible	I should see "text"
Verify text NOT visible	I should not see "text"
Verify checkbox checked	the "X" checkbox is checked
Verify radio selected	the "X" radio button is selected

FORM ACTION	WEBSHIP-JS STEP
Verify input value	I should see "X" value in the
	"Y" input element

Chapter Summary

- webship-js covers all standard form elements: text, select, checkbox, radio, file, rich text
- The `buildSelector` helper tries multiple strategies to find elements (label, placeholder, name, id)
- Use `wait for AJAX to finish` after actions that trigger server requests
- Scenario Outline is perfect for validation testing with many inputs
- Test the complete form workflow: fill → submit → verify
- Complex patterns (wizards, conditional fields) just combine basic steps
- Rich text editors need the special `fill the rich text editor field` step
- File uploads require test files in the `assets_folder`

Next chapter: mastering the art of waiting for dynamic content.

Chapter 7: Waiting and Dynamic Content

If there is one thing that separates reliable tests from flaky tests, it is how you handle timing. Modern web applications are asynchronous by nature. Content loads via AJAX. Animations play. WebSocket connections deliver real-time updates. If your test checks the page before the content arrives, it fails. Run it again, the server is faster, it passes. That is a flickering test, and it will destroy your team's confidence in the test suite.

This chapter teaches you how to handle every timing scenario you will encounter.

The BBR Smart-Wait Philosophy

Webship-js 2.0 follows a **Behaviour-Based Robotics** philosophy: **react to the environment, not the clock.** A test never sleeps for a fixed N seconds. Instead, every wait step returns as soon as the page is at the **edge** of activity, DOM ready, no in-flight network, no pending timers, no live mutations.

The runtime installs a small init script in every browser context. It exposes four signals:

SIGNAL	COUNTER / TIME	SOURCE
Fetch / XHR in flight	`window.__webshipAjaxCount`	wraps `window.fetch` and `XMLHttpRequest.send`
Pending `setTimeout`	`window.__webshipPendingTimers`	wraps `window.setTimeout` / `clearTimeout`
Last DOM mutation	`window.__webshipLastMutation`	`MutationObserver` on `<html>`
Network idle	(Playwright internal)	`page.waitForLoadState('networkidle')`

`smartSettle(page, budget)` polls all four atomically and returns when:

```
__webshipAjaxCount === 0
  && __webshipPendingTimers === 0
  && (Date.now() - __webshipLastMutation) >= 250 ms
```

…or when `budget` elapses, whichever comes first. **After every state-changing step** (click, press, fill, submit, select, check, uncheck, choose, attach, reload, navigate), webship-js silently runs `smartSettle(page, 1500)`. Tests rarely need an explicit wait between an action and its follow-up assertion. Disable per-run with `WEBSHIP_AUTO_SETTLE=off`.

The consequence: `wait 3s` no longer means “sleep three seconds.” It means “wait up to three seconds for the page to become quiet.” If the page quiets in 200ms, the wait exits in 200ms.

Understanding Asynchronous Web Applications

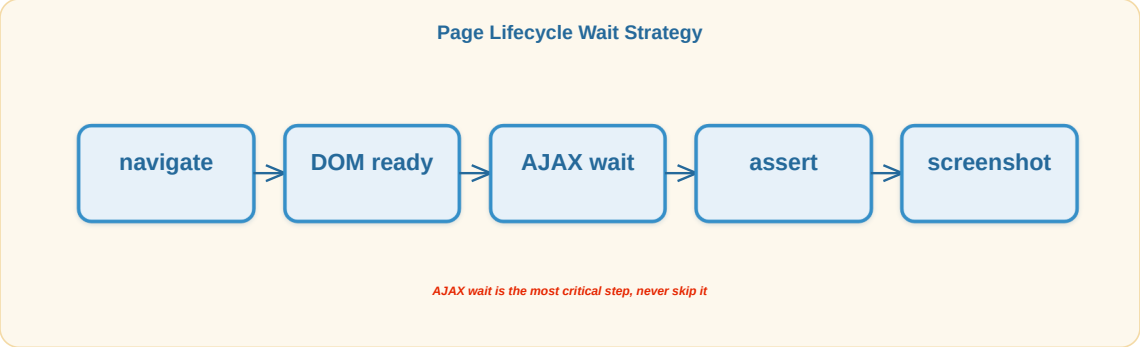


FIGURE 7.1 *Understanding Asynchronous Web Applications*

Before diving into recipes, let us understand what happens when a user clicks a button in a modern web app:

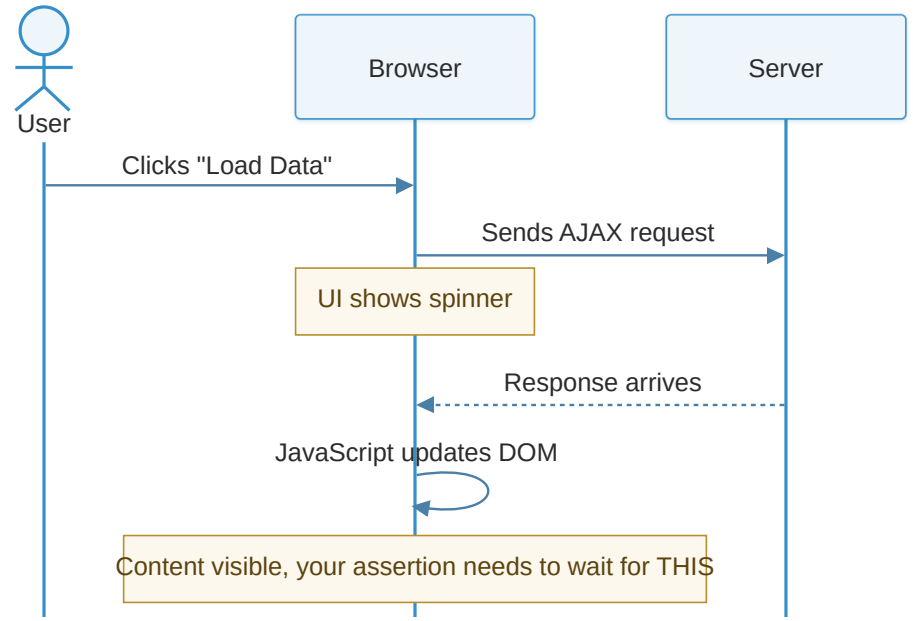


FIGURE 7.2 *Understanding Asynchronous Web Applications*

The entire cycle from click to visible content can take anywhere from 50ms to 10 seconds depending on the server, network, and client-side rendering. Your test needs to wait for the right moment.

Two Synchronization Strategies

There are two fundamental approaches to synchronization:

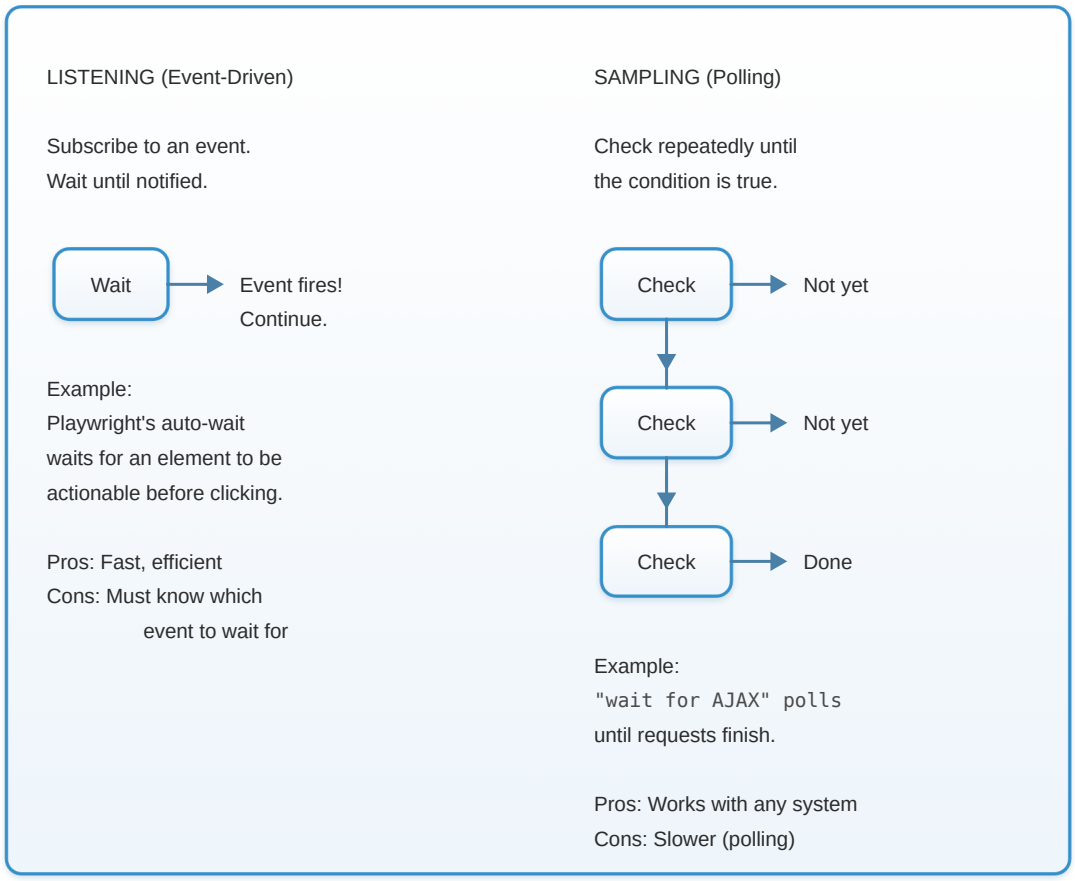


FIGURE 7.3 *Two Synchronization Strategies*

webship-js uses both strategies:

- **Playwright's auto-wait** (listening): automatically waits for elements to be visible and actionable before interacting
- `wait for AJAX to finish` (sampling): polls the browser to check if any network requests are still pending

Recipe 67 Wait for AJAX

Problem: The dashboard suite passes on your laptop every single time, then fails in CI on a different scenario every night. The pattern is always the same when you read the failure: the test pressed "Send", the staging server needed 800ms to answer, and the assertion ran at 300ms. Nobody broke the contact form, the test simply looked at the page before the response arrived. This is the single most common cause of flaky tests, and it has a one-line cure.

Solution:

```
Feature: AJAX Content Loading
  Scenario: Dashboard loads data asynchronously
    Given I am a logged in user with the "admin" user
    When I go to "/dashboard"
    And wait for AJAX to finish
    Then I should see "Statistics"
    And I should see "Recent Activity"

  Scenario: Form submission with AJAX
    Given I am on homepage
    When I go to "/contact"
    And I fill in "Name" with "Rajab"
    And I fill in "Message" with "Hello"
    And I press "Send"
    And wait for AJAX to finish
    Then I should see "Message sent"

  Scenario: Multiple AJAX calls in sequence
    Given I am a logged in user with the "admin" user
    When I go to "/admin/reports"
    And wait for AJAX to finish
    And I select "Last 30 days" from "Date Range"
    And wait for AJAX to finish
    And I press "Generate Report"
    And wait for AJAX to finish
    Then I should see "Report generated"
```

How It Works:

`And wait for AJAX to finish` implements the **sampling** strategy, but it samples something better than the network tab. webship-js wraps `window.fetch` and `XMLHttpRequest.send` on every page via `addInitScript`, keeping a live counter in `window.__webshipAjaxCount`, incremented when a request starts, decremented when it completes or fails. The wait step polls every 50ms until that counter reads zero **and** no wrapped `setTimeout` timers are pending (`window.__webshipPendingTimers`) **and** the DOM has been quiet for at least 250ms (`window.__webshipLastMutation`). So the step does not just wait for the response to come back, it waits for the JavaScript that *handles* the response, and the DOM updates that handler triggers, to finish too. That last part is what naive "wait for the response" approaches miss, and it is why your assertion finds "Message sent" actually rendered instead of merely fetched.

FIGURE 7.4 *Wait for AJAX (Recipe 67)*

When to use it:

Use `wait for AJAX to finish` after any action that triggers a server request:

- Form submissions
- Button clicks that load data
- Dropdown changes that fetch dependent data
- Navigation in SPAs
- Auto-save triggers
- Infinite scroll triggers

In practice you can often skip the explicit step, because webship-js runs `smartSettle(page, 1500)` automatically after every state-changing step, click, press, fill, select, the lot. Keep the explicit wait where the round trip matters: the AfterStep settle has a 1500ms budget, while the ex-

plicit step gets the full window, so a slow staging API outlasts the first but not the second. The explicit line also documents intent, six months from now, the reader of this scenario knows a server round trip happens here.

At scale: The duration-free design is what makes one feature file portable across twenty projects. The same `wait for AJAX to finish` absorbs a 50ms dev API and a 3-second staging API without edits, no per-project timeout tuning, no `wait 2s` "because project B is slow" creeping into a shared suite. When someone proposes adding a sleep to fix a flaky scenario, point them here first; if the scenario still flakes behind this step, the finding is real and belongs in a bug ticket, not a longer sleep.

Recipe 68 Wait for Page Load

Problem: The analytics page pulls in two chart libraries, a map widget, and forty thumbnails. On your machine it renders in under a second; on the shared CI runner, throttled, cold cache, three other suites competing for bandwidth, the chart script is still downloading when your assertion fires. `wait for AJAX to finish` does not help here, because nothing is an XHR yet: it is plain old resources arriving slowly, before the page's JavaScript even gets a chance to run.

Solution:

```
Feature: Heavy Page Loading
  Scenario: Report page with many charts
    Given I am a logged in user with the "admin" user
    When I go to "/admin/analytics"
    And wait for the page to load
    Then I should see "Analytics Dashboard"
    And I should see "Traffic Chart"
    And I should see "Revenue Chart"
```

How It Works:

`And wait for the page to load` checks `document.readyState` until it equals `"complete"`. This means:

- The HTML document has been fully parsed
- All sub-resources (images, stylesheets, scripts) have finished loading
- The `load` event has fired

This is more thorough than `wait for AJAX` because it waits for ALL resources, not just XHR/Fetch requests.

Know what it does *not* cover, though: anything the page's JavaScript starts **after** the `load` event. A dashboard that fires its data requests on load will be "complete" by this step's definition while the charts are still empty. That is why the two waits pair so well, `wait for the page to load` for the resource phase, `wait for AJAX to finish` for the data phase. Also worth knowing: every navigation step already probes `domcontentloaded` and `networkidle` inside its `smartSettle` budget, so ordinary pages rarely need this step at all. Reach for it on the genuinely heavy ones, report pages, media galleries, anything where the resources themselves are the slow part.

Recipe 69 Explicit Timed Waits

Problem: A carousel advances every three seconds. A toast dismisses itself after five. These are the rare cases where time *is* the behaviour, the acceptance criterion literally says "the message disappears after 5 seconds", and the only honest way to test it is to let those seconds pass. The danger is that timed waits are habit-forming: the same step that legitimately tests a slideshow becomes the duct tape a tired engineer reaches for at 6 p.m. when a flaky scenario needed an AJAX wait instead. Know exactly when this step is the right tool, and say no the rest of the time.

Solution:

```
Feature: Timed Waits
  Scenario: Wait for slideshow transition
    Given I am on homepage
    And wait 3s
    Then I should see "Slide 2"
    And wait 3s
    Then I should see "Slide 3"

  Scenario: Wait for notification to auto-dismiss
    Given I am a logged in user with the "admin" user
    When I go to "/settings"
    And I press "Save"
    And wait for AJAX to finish
    Then I should see "Settings saved"
    # Wait for the toast notification to auto-dismiss
    And wait 5s
    Then I should not see "Settings saved"

  Scenario: Default wait
    Given I am on homepage
    When I press "Process"
    And wait
    Then I should see "Done"
```

How It Works:

- `And wait 2s`, waits **up to** 2 seconds for the page to settle
- `And wait 3s`, waits up to 3 seconds
- `And wait 5s`, waits up to 5 seconds
- `And wait`, uses the default budget configured in `minWaitTime`

Notice the "up to". In webship-js 2.0 these are not raw sleeps, remember, there are no static sleeps anywhere in the framework. Each duration becomes a `smartSettle` budget. If the page is already quiet, the wait exits early instead of burning your CI minutes. If a timer is pending, the carousel's next-slide callback, the toast's dismiss timer, both ordinary `setTimeout` calls counted

in `window.__webshipPendingTimers`, the settle condition cannot be met, so the wait holds for its full budget. That gives you exactly the behaviour each scenario above needs: sleep-like patience while the animation timer runs, zero wasted time when there is nothing to wait for.

When timed waits are appropriate:

USE timed waits for:

✓ CSS animations with known duration
✓ Auto-dismiss notifications
✓ Slideshow transitions
✓ Debounced input events
✓ Rate-limited operations

RULE: If the wait time is part of the FEATURE (animation is 3 seconds), test with a timed wait.

AVOID timed waits for:

✗ AJAX responses (use `"wait for AJAX" instead)`
✗ Page loads (use "wait for the page to load" instead)
✗ "Just in case" waits
✗ Replacing actual sync

RULE: If you are waiting for an operation to complete, use event-based waiting instead.

FIGURE 7.5 *Explicit Timed Waits (Recipe 69)*

At scale: Do the multiplication before you approve a timed wait in review. A `wait 5s` in a Background runs once per scenario, two hundred scenarios in the suite, forty suites in the nightly farm, and that single line is buying minutes of runner time per night, every night. A convention that has paid for itself on every portfolio I have worked with: every `wait Ns` carries a comment naming the timer or animation it covers (`# toast auto-dismisses after 5s`). If nobody can name the thing being waited for, the wait gets deleted, it was a sleep wearing a costume.

Recipe 70 Handling Race Conditions

Problem: “Place Order” is never one operation. The server validates, charges the card, creates the order, queues the email, and the client repaints two or three times along the way. Your assertion fires in the gap between the response arriving and the confirmation actually rendering, and the failure only reproduces under CI load, never on your laptop while you sit there watching it pass. These between-the-operations failures are the ones that teach teams to distrust their own suite.

Solution:

```
Feature: Complex Operations
  Scenario: Order processing with multiple steps
    Given I am a logged in user with the "customer" user
    When I go to "/cart"
    And I press "Place Order"
    And wait for AJAX to finish
    # Server processes: validate → charge → create order → send email
    # The AJAX response comes back, but the page might still be updating
    And wait 1s
    Then I should see "Order Confirmed"
    And I should see "Order #"
```

How It Works:

In older frameworks the double wait was mandatory: the AJAX wait covered the network, and a sleep covered the rendering. webship-js 2.0 closes most of that gap on its own, `wait for AJAX to finish` already refuses to return until the DOM has been quiet for 250ms and no timers are pending, so the response *handler's* work is normally covered. What the trailing `wait 1s` buys you is the work that starts **after** a quiet window: a confirmation screen that fires a follow-up request once it mounts, a staged animation that kicks off a beat later. Because `wait 1s` is itself a bounded settle (Recipe 69), it costs almost nothing when the page stays quiet, and if late work does show up, it holds until that work finishes or the budget runs out. The "double wait" pattern handles:

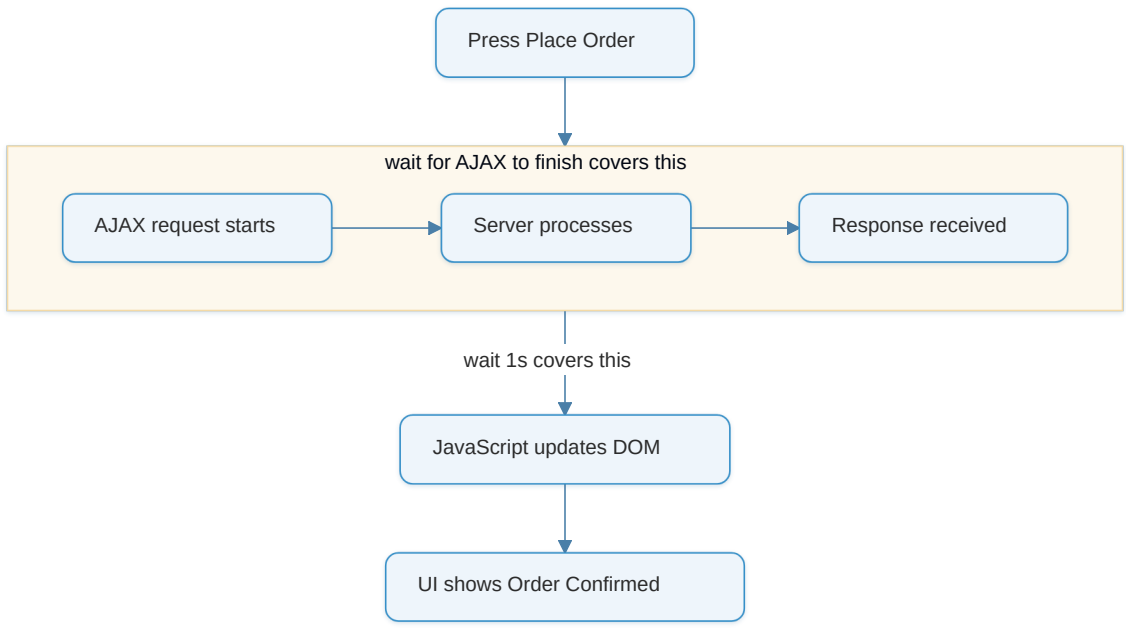


FIGURE 7.6 *Handling Race Conditions (Recipe 70)*

One honest rule to go with it: if a scenario needs more than one of these trailing waits, stop adding waits. A page that takes multiple settle budgets to reach its final state after one click has a real responsiveness problem, the kind your users feel as "did my order go through?", and that belongs in a bug ticket, not in a longer test.

Recipe 71 Testing Infinite Scroll

Problem: Infinite scroll always works for the first batch, that is the batch everyone looks at. The bugs live further down: batch two renders the same ten posts as batch one, loading silently stops after the third fetch, or a fast scroll fires the load trigger twice and duplicates appear. Your manual testers rarely scroll past the second batch, and your users find out on the train, on a phone, three weeks after the release. The test suite should be the one reader who always scrolls.

Solution:

```
Feature: Infinite Scroll
  As a user browsing a long feed
  I want content to load as I scroll
  So that I can browse without clicking "next page"

  Scenario: Initial page shows first batch
    Given I am on homepage
    When I go to "/feed"
    And wait for AJAX to finish
    Then I should see "Post 1"

  Scenario: Scrolling loads more content
    Given I am on homepage
    When I go to "/feed"
    And wait for AJAX to finish
    Then I should see "Post 1"

    When I scroll to bottom
    And wait for AJAX to finish
    Then I should see "Post 11"

    When I scroll to bottom
    And wait for AJAX to finish
    Then I should see "Post 21"
```

How It Works:

`When I scroll to bottom` scrolls the window to the very bottom of the document, which carries it past whatever sentinel the application watches, whether that is a scroll-position listener or an `IntersectionObserver` on a "load more" marker. The fetch that fires gets counted in `window.__webshipAjaxCount`, so the `wait for AJAX to finish` that follows holds until the new posts are not just downloaded but rendered (DOM quiet for 250ms).

The assertions are doing careful work here. "Post 11" and "Post 21" can only exist in the second and third batches, so each scroll proves a *new* batch arrived, asserting "Post 1" again would pass even if scrolling did nothing. If your feed has a duplicate-batch bug to guard against, add a count assertion: `Then ".post" should have a count of 20` after the first scroll pins down that batch two added exactly ten items rather than re-appending batch one.

At scale: Two assumptions are baked into this scenario, and both break when the feature file travels. First, "ten posts per batch" is seed data, on the project with seven posts total there is no Post 11, so pair scroll scenarios with seeded fixtures or keep the batch size in one shared place. Second, the trigger depends on viewport height: a 1920x1080 runner shows more posts up front than a laptop window, and on a short content set the "second batch" may already be loaded before anyone scrolls. Pin the viewport for feed suites, or assert structurally (the count step) instead of on specific item names.

Recipe 72 Testing Loading Spinners

Problem: The support ticket reads "it just spins forever." A failed request left the spinner up with no error message, and the user stared at it until they gave up. Your suite never noticed, because every scenario asserted the data and nobody ever asserted that the *loading state* resolves. A spinner that never leaves is an outage the page hides from your monitoring, the HTTP responses look fine, the page "loaded", and the user sees an animation going nowhere.

Solution:

```
Feature: Loading States
  Scenario: Loading spinner appears and disappears
    Given I am a logged in user with the "admin" user
    When I go to "/dashboard"
    # The spinner might be visible briefly
    And wait for AJAX to finish
    # After AJAX completes, spinner should be gone and content should be
visible
    Then I should see "Dashboard"
    And I should not see "Loading..."
```

How It Works:

The two assertions together pin the end state: the content arrived **and** the loading indicator left. Either one alone is weaker than it looks, “Dashboard” can render in a header while a panel spinner sits stuck below it, and “no spinner” is trivially true on a page that failed before showing one.

Notice what the recipe does *not* assert: that the spinner appeared. That check is inherently racy, on a fast response the spinner mounts and unmounts in under 100ms, so a `Then I should see "Loading..."` placed after the navigation fails precisely when your app is healthy and quick. Assert the destination, not the journey. If the spinner is an element rather than text, the web-first form is sharper: `Then ".spinner" should be hidden` polls every 100ms for up to 5 seconds, so it tolerates a spinner that lingers through a fade-out animation.

When the stuck-spinner bug does ship, this scenario fails with “Loading…” still on the page, and because webship-js captures a screenshot on failure by default (`WEBSHIP_SCREENSHOT_ON_FAILED` is true out of the box), the report shows the frozen spinner. That artifact turns “works for me” into a two-minute diagnosis: the reviewer sees the spinner, checks which request never resolved, done.

Recipe 73 Testing Auto-Refresh Content

Problem: The monitoring dashboard refreshes itself every ten seconds, until a deploy quietly breaks the refresh loop, and the operations team spends a day staring at numbers from 9:14 a.m. without knowing it. Auto-refresh is the feature whose failure looks exactly like success: the page renders, the data is plausible, and nothing on screen admits it stopped updating. A test has to do what no human does deliberately, load the page, wait out a full refresh interval, and check that the page is still alive afterwards.

Solution:

```
Feature: Auto-Refresh
  Scenario: Dashboard data refreshes automatically
    Given I am a logged in user with the "admin" user
    When I go to "/monitoring"
    And wait for AJAX to finish
    Then I should see "Server Status"

    # Wait for the auto-refresh interval (e.g., 10 seconds)
    And wait 10s
    And wait for AJAX to finish
    # Content should still be present (and potentially updated)
    Then I should see "Server Status"
    And I should see "Last Updated"
```

How It Works:

The `wait 10s` gives the refresh interval room to elapse. Because the dashboard's pending refresh keeps the page from ever settling fully, the wait holds for its whole budget, which is the point: you want the interval to fire *during* the wait. The `wait for AJAX to finish` that follows catches the refresh request if it is still mid-flight, so the assertions run against the post-refresh page, not a half-painted one.

Be honest about what this scenario proves, though. `I should see "Server Status"` would also pass if the refresh never fired, the original render says the same thing. To prove the refresh actually *happened*, assert on something only the refresh produces: a "Last Updated" timestamp that

the page rewrites on every cycle, a counter that increments, a request you can observe. The strongest version pairs this with network recording, `Given I start recording network requests` before the wait, then assert the refresh endpoint was called, so the evidence is the request itself, not a lookalike pixel.

And for long intervals, stop paying for real time at all: webship-js ships clock steps that install and advance Playwright's `page.clock`. A dashboard that refreshes every five minutes becomes testable in milliseconds, install the fake clock, advance it five minutes, assert the refresh fired. Chapter 38's clock recipes cover the pattern.

At scale: Real-time dashboards are where nightly CI bills quietly grow. Every literal `wait 10s` runs on every project, every night, forever, forty suites with three refresh scenarios each is twenty minutes of runners doing nothing but letting intervals tick. Across a portfolio, make the clock steps the default for anything with an interval longer than a couple of seconds, and reserve real-time waits for one smoke scenario per dashboard that proves the production timer is actually wired up.

The Anti-Pattern: Sleep-Driven Testing

This is what NOT to do:

```
# BAD: Sleep everywhere
Scenario: The slow test
  Given I am on homepage
  And wait 3s                    # Why?
  When I go to "/login"
  And wait 2s                    # Why?
  And I fill in "Email" with "a@b.c"
  And wait 1s                    # Why?
  And I press "Login"
  And wait 5s                    # Why?
  Then I should see "Dashboard"
```

Every one of those waits answers “why?” with “just in case.” Under webship-js 2.0 they are settle budgets rather than raw sleeps, so a quiet page lets them exit early, but the moment the page has any pending timer (an animation library, an analytics retry loop, a carousel), each one burns its full budget. That is up to 11 seconds per run for zero added confidence, and worse: the waits paper over whatever timing problem prompted them, so the underlying flake is still in there, waiting for a slower runner. The correct version:

```
# GOOD: Wait only when needed
Scenario: The fast test
  Given I am on homepage
  When I go to "/login"
  And I fill in "Email" with "a@b.c"
  And I press "Login"
  And wait for AJAX to finish
  Then I should see "Dashboard"
```

Same test. Fraction of the time. More reliable because it waits for the actual event (AJAX completion) instead of an arbitrary duration.

At scale: Sleep-driven style spreads by copy-paste. The first padded scenario becomes the template for the next fifty, then the suite gets cloned to the next project with the padding intact, and a year later the nightly farm spends more wall-clock time sleeping than testing. Make “every wait names its reason” a review rule across all projects, and audit periodically: search the portfolio’s feature files for `wait \d` and challenge each hit. On one distribution we maintained across dozens of sites, that single audit recovered more CI time than any parallelization work we did.

Wait Strategy Decision Tree

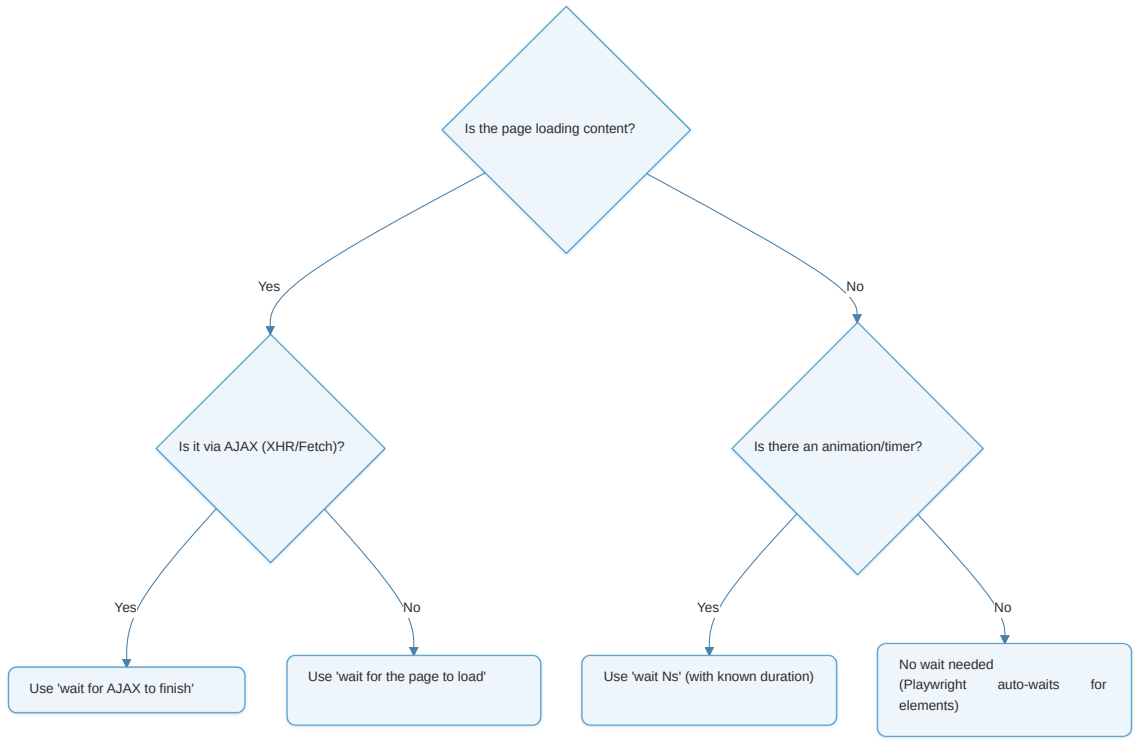


FIGURE 7.7 *Wait Strategy Decision Tree*

Chapter Summary

- Timing is the #1 cause of flaky tests
- Two strategies: **Listening** (event-driven, fast) and **Sampling** (polling, simple)
- `wait for AJAX to finish` handles most dynamic content scenarios
- `wait for the page to load` handles resource-heavy pages
- `wait Ns` is for animations and timers with known durations, and in 2.0 it is a settle *budget*, not a raw sleep: quiet pages exit early
- `wait` (no duration) uses the configured `minWaitTime` default
- Avoid arbitrary sleeps, they waste time and do not actually fix timing issues
- The “double wait” pattern (AJAX + short timed wait) handles complex operations
- Playwright auto-waits for elements to be actionable, you do not need waits before `fill in` or `press`

Next: Modal and dialog testing recipes.

Chapter 8: Modal and Dialog Testing

Modal dialogs are everywhere in modern web applications. Confirmation prompts before deleting a record. Login forms that slide down from the top. Cookie consent banners. GDPR notifications. Product quick-view overlays. Every one of these presents a unique challenge for automated tests because modals exist as a layer on top of the page, they appear, they capture focus, they block interaction with the content behind them, and they disappear.

If your test does not account for the modal lifecycle, waiting for it to appear, interacting with it, and waiting for it to close, you will get race conditions, missed clicks, and tests that pass locally but fail in CI.

This chapter gives you recipes for every modal pattern you will encounter.

Understanding Modal Architecture

Before we dive into recipes, it helps to understand how modal dialogs work at the HTML level:

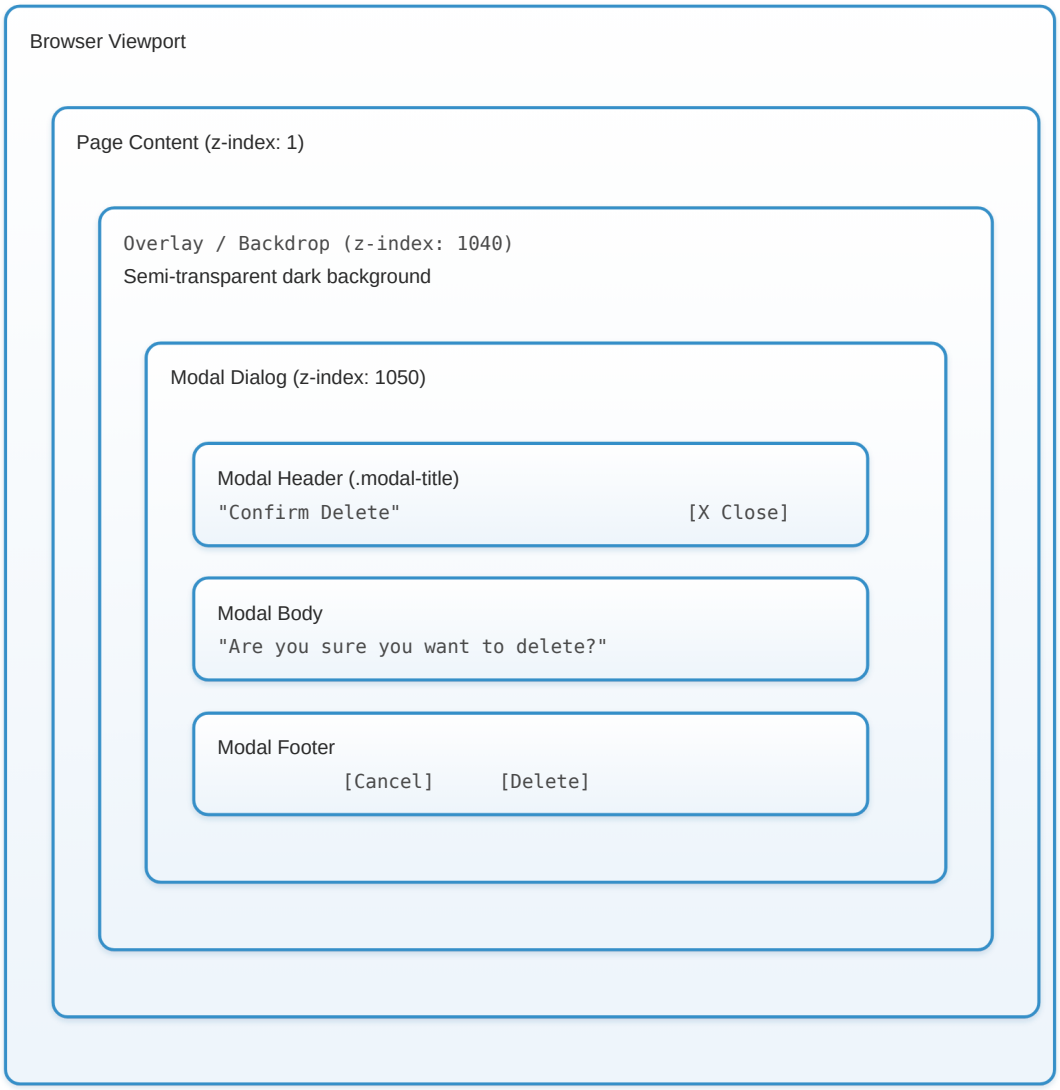


FIGURE 8.1 *Understanding Modal Architecture*

There are two fundamentally different kinds of dialogs:

NATIVE BROWSER DIALOGS	CUSTOM MODAL DIALOGS
Created by JavaScript:	Created by HTML/CSS/JS:
• alert("message")	• Bootstrap .modal
• confirm("question?")	• Material UI Dialog
• prompt("enter value:")	• Custom [role="dialog"]
	• Any styled as modal
Behavior:	
• Blocks ALL JavaScript	Behavior:
• Cannot be styled	• Does NOT block JS
• Cannot be inspected in DOM	• Fully styleable
• Handled via Playwright's	• Part of the DOM
page.on('dialog') event	• Found with CSS selectors
Testing approach:	Testing approach:
Set up listener BEFORE the	Wait for modal to appear,
action that triggers it.	interact with it, wait for
	it to close.

webship-js provides built-in steps for **custom modal dialogs** (the second type). For native browser dialogs, you would write custom step definitions using Playwright’s dialog event API. Most modern applications use custom modals rather than native dialogs.

Recipe 74 Testing Modal Open and Close

Problem: The delete-confirmation modal takes 300ms to fade in. Your test presses “Delete Product” and asserts on the modal text immediately, and on your laptop it passes, because your laptop is fast. On the CI runner the assertion fires mid-fade and the scenario goes red, with a failure screenshot showing the modal sitting there, perfectly open, mocking you. This is usually a team’s first encounter with modal timing, and the new hire who hits it on their first PR will conclude the suite is haunted unless someone hands them the lifecycle steps.

Solution:

```
Feature: Modal Dialog Open/Close
  As a user
  I want modal dialogs to open and close correctly
  So that I can interact with overlaid content

  Scenario: Modal opens when button is clicked
    Given I am on homepage
    When I go to "/products"
    And I press "Delete Product"
    And wait for the modal to appear
    Then I should see a modal dialog
    And I should see "Confirm Delete" in the modal

  Scenario: Modal closes with close button
    Given I am on homepage
    When I go to "/products"
    And I press "Delete Product"
    And wait for the modal to appear
    Then I should see a modal dialog
    When I close the modal
    And wait for the modal to disappear
    Then I should not see a modal dialog

  Scenario: Modal closes with Escape key
    Given I am on homepage
    When I go to "/products"
    And I press "Delete Product"
    And wait for the modal to appear
    Then I should see a modal dialog
    When I dismiss the modal
    And wait for the modal to disappear
    Then I should not see a modal dialog
```

How It Works:

The modal lifecycle in webship-js follows this pattern:

1. TRIGGER		2. WAIT		3. VERIFY
Press button or	──►	wait for the modal	──►	should see a modal
click link that		to appear		dialog
opens the modal				
4. INTERACT		5. CLOSE		6. VERIFY CLOSED
Click buttons in	──►	close the modal	──►	should not see a
the modal, fill		OR		modal dialog
forms, read text		dismiss the modal		

webship-js detects modals using a broad CSS selector that covers popular frameworks:

```
.modal, .modal.show, .modal.in, [role="dialog"], .dialog, .popup
```

This means it works with Bootstrap, Material UI, Foundation, custom dialogs, and any element with `role="dialog"`.

The `close the modal` step first looks for a close button (`.close`, `.btn-close`, `[aria-label="Close"]`). If no close button is found, it sends the Escape key as a fallback. `dismiss the modal` goes straight to Escape, which is why the third scenario exists: Escape-to-close is what keyboard users rely on, and a modal that traps them is both a bug and an accessibility failure.

Why a dedicated wait instead of `wait for AJAX to finish`? Because opening a modal usually involves zero network traffic, it is a class flip plus a CSS transition, and a transition is compositor work that the AJAX counter and timer wrapper never see. The mutation observer registers the class change as a single blip, so `smartSettle` can declare the page quiet while the modal is still fading in. The modal wait polls for a *visible* match of the modal selector instead, which is the one

signal that actually means “the user can see this now.” The same logic applies in reverse on close: `wait for the modal to disappear` holds until no visible modal remains, so your next click cannot land on a dying backdrop.

At scale: The broad selector is what lets one set of modal steps serve an entire portfolio, the Bootstrap project, the Material UI project, and the hand-rolled design system all match, as long as they use standard classes or `role="dialog"`. The project that breaks the pattern is the one with a bare `<div class="overlay">` and no role attribute. Resist the urge to fork the suite for it: file the missing `role="dialog"` as an accessibility bug instead. The fix makes the shared steps work, makes screen readers work, and costs the frontend team one attribute.

Recipe 75 Verifying Modal Content

Problem: An admin clicks Delete on row three, the modal asks “Are you sure you want to delete this user?”, they click Yes, and row *five’s* account dies, because a stale index bound the modal to the wrong record. “A modal appeared” was never the test. The test is: the modal is talking about the right record, by name, with the right consequences spelled out. Generic confirmation modals that never mention what they are about to destroy are how wrong-row deletions reach production.

Solution:

```
Feature: Modal Content Verification
  Scenario: Modal shows correct title
    Given I am a logged in user with the "admin" user
    When I go to "/users"
    And I press "Add User"
    And wait for the modal to appear
    Then I should see a modal with title "Add New User"

  Scenario: Modal shows correct body text
    Given I am a logged in user with the "admin" user
    When I go to "/users"
    And I click "Delete" in the "john@example.com" row
    And wait for the modal to appear
    Then I should see "Are you sure you want to delete this user?" in the
modal
    And I should see "john@example.com" in the modal
    And I should see "This action cannot be undone" in the modal

  Scenario: Modal shows dynamic data
    Given I am a logged in user with the "admin" user
    When I go to "/orders"
    And I click "View Details" in the "ORD-12345" row
    And wait for the modal to appear
    Then I should see "Order Details" in the modal
    And I should see "ORD-12345" in the modal
    And I should see "Total: $149.99" in the modal
```

How It Works:

webship-js provides three distinct modal content assertions:

STEP	WHAT IT CHECKS
I should see a modal dialog	Any modal is visible
I should see a modal with title "X"	Modal title heading
I should see "X" in the modal	Text anywhere in modal
I should see the "login" modal	Modal by ID/class/data
I should not see a modal dialog	No modal is visible
I should not see "X" in the modal	Text NOT in modal

The title check (`modal with title "X"`) searches for text in `.modal-title` , `.dialog-title` , `h1` , `h2` , `h3` , `[title]` , and `[aria-label]` elements within the modal. This covers virtually every framework's title pattern.

The scoping is the part that earns its keep. A plain `Then I should see "john@example.com"` searches the whole page, and john@example.com is sitting right there in the user table *behind* the modal, so the page-level assertion passes even if the modal bound to nothing at all. `in the modal` restricts the search to the visible modal element, which means the assertion can only pass if the modal itself carries the record's identity. That one scoped line is the difference between testing "a dialog opened" and testing "the dialog knows which user it is about to delete." The same applies to the order-details scenario: "ORD-12345" appears in the orders list too; only the scoped check proves the modal fetched the right order.

Recipe 76 Clicking Buttons Inside Modals

Problem: The products table has a Delete button on every row, twenty of them, and the confirmation modal adds a twenty-first. An unscoped click step now has twenty-one candidates: it can pick a row button that the backdrop blocks and time out, or worse, find one that is still clickable and delete something you never asked about. The Tuesday before release is not the day to discover your test has been confirming the wrong Delete. Inside a modal, every click needs to stay inside the modal.

Solution:

```
Feature: Modal Button Interactions
  Scenario: Confirm a delete action
    Given I am a logged in user with the "admin" user
    When I go to "/products"
    And I click "Delete" in the "Laptop Pro" row
    And wait for the modal to appear
    Then I should see "Are you sure?" in the modal
    When I click "Delete" in the modal
    And wait for AJAX to finish
    Then I should not see a modal dialog
    And I should see "Product deleted"
    And I should not see "Laptop Pro"

  Scenario: Cancel a delete action
    Given I am a logged in user with the "admin" user
    When I go to "/products"
    And I click "Delete" in the "Laptop Pro" row
    And wait for the modal to appear
    Then I should see "Are you sure?" in the modal
    When I click "Cancel" in the modal
    And wait for the modal to disappear
    Then I should not see a modal dialog
    And I should see "Laptop Pro"

  Scenario: Save form inside modal
    Given I am a logged in user with the "admin" user
    When I go to "/categories"
    And I press "Add Category"
    And wait for the modal to appear
    And I fill in "Category Name" with "Electronics"
    And I fill in "Description" with "Electronic devices and accessories"
    When I click "Save" in the modal
    And wait for AJAX to finish
    Then I should not see a modal dialog
    And I should see "Category created"
    And I should see "Electronics"
```

How It Works:

`When I click "Delete" in the modal` does the following:

1. Waits for a modal to become visible (up to 10 seconds)
2. Finds the first visible modal using the `MODAL_SELECTOR`
3. Within that modal, looks for buttons, links, or elements with `role="button"` that match the text
4. Clicks the matching element

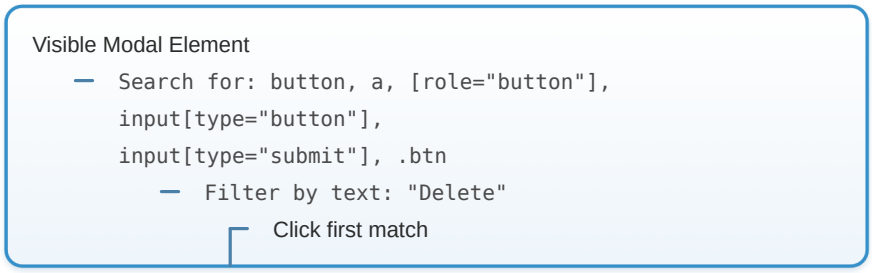


FIGURE 8.2 *Clicking Buttons Inside Modals (Recipe 76)*

Important: The step says `click "X" in the modal`, not `press "X"`. This is because the step searches for any clickable element (buttons AND links) within the modal context. Outside of modals, `click` is for links and `press` is for buttons, but inside modals, `click "X" in the modal` handles both.

The scoping solves the twenty-one-Delete-buttons problem by construction: the search never leaves the visible modal's subtree, so the row buttons behind the backdrop are not candidates at all. And because the step waits up to 10 seconds for a visible modal before searching, it absorbs the fade-in race from Recipe 74 on its own, though keeping the explicit `wait for the modal to appear` line makes the scenario read in the order things actually happen, which your reviewers will thank you for.

Look at what the first two scenarios assert *after* the click, too. Confirm-then-deleted and cancel-then-still-there are assertions on the **data outcome**, not just on the modal closing. A modal can close politely and still have done the wrong thing; "I should not see Laptop Pro" is what catches that.

At scale: Make it a written convention: inside a modal, only the scoped step, never a bare `click` or `press`, even when it happens to work today. It happens to work right up until a project adds one more button with the same caption somewhere on the page, and then a suite that passed for a year starts failing in a way that looks random. On a portfolio of suites maintained by rotating engineers, this is exactly the class of rule that belongs in the team's conventions doc and in code review, because the failure it prevents is expensive to diagnose and trivial to avoid.

Recipe 77 Modal Form Submission

Problem: Admin UIs love modal forms, add a user without ever leaving the list. The bug that slips through looks like this: the admin clicks Create User, the modal closes, the toast says nothing, and the list never gains a row, because the POST failed and the modal closed anyway. Or its mirror image: validation errors render inside the modal while your test asserted on page-level text and called it a pass. Modal forms need the form checks from Chapter 6 *and* the modal lifecycle from this chapter, in the same scenario.

Solution:

```
Feature: Modal Forms
  As an admin
  I want to add and edit records using modal forms
  So that I can manage content without leaving the page

  Scenario: Add a new user via modal form
    Given I am a logged in user with the "admin" user
    When I go to "/admin/users"
    And I press "Add User"
    And wait for the modal to appear
    Then I should see a modal with title "Add New User"
```

```
    When I fill in "First Name" with "Jane"
    And I fill in "Last Name" with "Doe"
    And I fill in "Email" with "jane@example.com"
    And I select "Editor" from "Role"
    And I click "Create User" in the modal
    And wait for AJAX to finish
    Then I should not see a modal dialog
    And I should see "User created successfully"
    And I should see "jane@example.com"

  Scenario: Edit a record via modal form
    Given I am a logged in user with the "admin" user
    When I go to "/admin/users"
    And I click "Edit" in the "jane@example.com" row
    And wait for the modal to appear
    Then I should see a modal with title "Edit User"
    When I fill in "First Name" with "Janet"
    And I click "Save Changes" in the modal
    And wait for AJAX to finish
    Then I should not see a modal dialog
    And I should see "User updated"
    And I should see "Janet"

  Scenario: Modal form validation
    Given I am a logged in user with the "admin" user
    When I go to "/admin/users"
    And I press "Add User"
    And wait for the modal to appear
    When I click "Create User" in the modal
    # Form should show validation errors inside the modal
    Then I should see "First Name is required" in the modal
    And I should see "Email is required" in the modal
    # Modal should stay open
    And I should see a modal dialog
```

How It Works:

Modal forms work seamlessly with webship-js because form steps (`fill in`, `select from`, `check`) operate on the entire page DOM. Since the modal is part of the DOM and its form fields are visible, the standard form steps find and fill them.

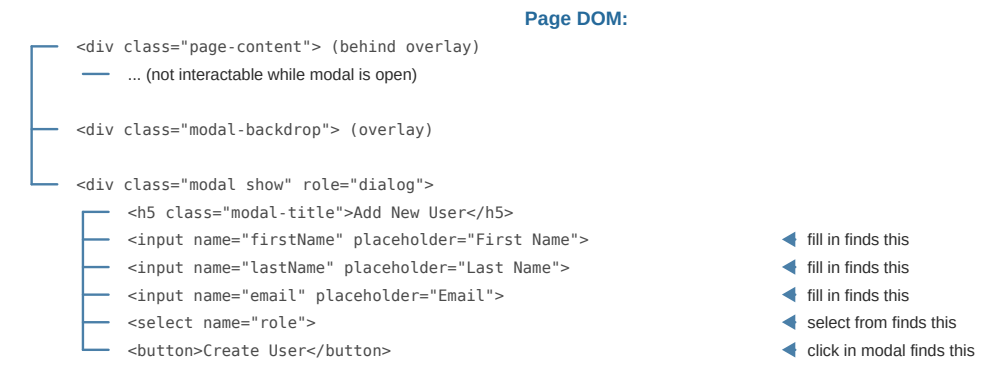


FIGURE 8.3 *Modal Form Submission (Recipe 77)*

Note: If the page behind the modal has fields with the same labels, Playwright will interact with the **visible, interactable** one, which is the modal's field since the backdrop blocks the page fields. This is correct behavior.

The field resolution is the same label-first strategy from Chapter 6, label, then id, name, placeholder, so a modal form scenario reads exactly like a page form scenario plus the lifecycle steps. Two assertions carry most of the weight here. In the happy-path scenarios, `Then I should not see a modal dialog` followed by the new record's email proves the full round trip: submitted, accepted, closed, *and* the list refreshed with real data. In the validation scenario, the closing `And I should see a modal dialog` is the one teams forget, it pins that the modal **stayed open** on invalid input. A modal that closes on a failed submit throws away everything the user typed, which is the kind of bug that never shows in a happy-path-only suite.

At scale: Add-edit-via-modal is the same workflow on every admin UI you will ever test, which makes this scenario shape a template worth standardizing across projects: open, assert title, fill, submit, assert closed, assert the record. The validation variant doubles as an acceptance criterion product owners write naturally, "if required fields are empty, the dialog stays open and says what is missing", and when the scenario in the suite is that sentence verbatim, sign-off stops being an act of trust and becomes reading a green report.

Recipe 78 Testing Confirmation Dialogs

Problem: Every destructive action in your app is one click away from a support escalation, and the confirmation modal is the seatbelt. The failure modes are nastier than “no modal”: there is the modal that deletes even when you click Cancel, the X button that confirms instead of closing, and the bulk dialog that says “Delete 5 posts?” while 7 are selected. Teams reliably test the confirm path and skip the cancel path, and the cancel path is the one protecting data someone cares about. Production records pay for that gap.

Solution:

```
Feature: Confirmation Dialogs
  As a user
  I want to confirm before destructive actions
  So that I do not accidentally lose data

  Background:
    Given I am a logged in user with the "admin" user

  Scenario: Confirm and proceed with deletion
    When I go to "/admin/posts"
    And I click "Delete" in the "Draft Post" row
    And wait for the modal to appear
    Then I should see "Confirm Delete" in the modal
    And I should see "Are you sure you want to delete" in the modal
    When I click "Yes, Delete" in the modal
    And wait for AJAX to finish
    Then I should not see a modal dialog
    And I should not see "Draft Post"

  Scenario: Cancel the deletion
    When I go to "/admin/posts"
    And I click "Delete" in the "Important Post" row
    And wait for the modal to appear
    Then I should see "Confirm Delete" in the modal
    When I click "Cancel" in the modal
    And wait for the modal to disappear
```

```
    Then I should not see a modal dialog
    And I should see "Important Post"

  Scenario: Close confirmation with X button
    When I go to "/admin/posts"
    And I click "Delete" in the "Important Post" row
    And wait for the modal to appear
    When I close the modal
    And wait for the modal to disappear
    Then I should see "Important Post"

  Scenario: Bulk delete with confirmation
    When I go to "/admin/posts"
    And I check "Select All"
    And I press "Delete Selected"
    And wait for the modal to appear
    Then I should see "Delete 5 posts?" in the modal
    When I click "Delete All" in the modal
    And wait for AJAX to finish
    Then I should not see a modal dialog
    And I should see "5 posts deleted"
```

The Confirmation Dialog Pattern:

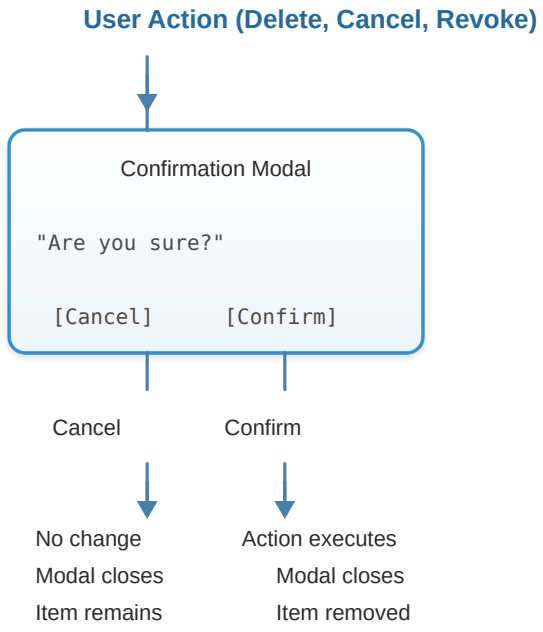


FIGURE 8.4 *Testing Confirmation Dialogs (Recipe 78)*

This is one of the most common modal patterns. The key to testing it is:

1. Trigger the action
2. Wait for the modal to appear
3. Verify the warning message is correct
4. Click the appropriate button
5. Verify the outcome

Step 5 is where these scenarios get their teeth. The cancel and X-button scenarios end with `And I should see "Important Post"`, an assertion that the record **survived**, not merely that the modal left the screen. A dialog can close gracefully and still have fired the delete; only the data outcome tells you which path the code really took. The bulk scenario adds one more pin: “Delete 5 posts?” verifies the modal binds to the actual selection count, which catches the off-by-one and stale-selection bugs that make bulk operations the most feared button in any admin panel.

At scale: Destructive-action scenarios deserve their own tag across the portfolio, `@destructive` works, and a hard rule about where they run: seeded, disposable data only, never an environment whose contents anyone would miss. With the tag in place, your pipeline can run them on every project against fresh fixtures, and your delivery managers get a sentence they can actually use in a release review: every delete in every product asks first, and cancel means cancel. The cancel-path scenario is the cheapest data-loss insurance your team will ever write.

Recipe 79 Testing Modal with Specific Identifier

Problem: One page, four possible modals: login, signup, password reset, session-expiry warning. "A modal appeared" is true even when the *wrong* one appears, a routing mix-up shows the signup form to someone who pressed Sign In, and your generic assertion waves it through. The user lands somewhere confusing; your report stays green. When a page can produce more than one dialog, the test has to name the one it expects.

Solution:

```
Feature: Specific Modal Identification
  Scenario: Login modal appears
    Given I am on homepage
    When I press "Sign In"
    And wait for the modal to appear
    Then I should see the "login" modal
    And I should see "Email" in the modal
    And I should see "Password" in the modal

  Scenario: Settings modal appears
    Given I am a logged in user with the "admin" user
    When I go to "/dashboard"
    And I press "Quick Settings"
    And wait for the modal to appear
    Then I should see the "settings" modal
    And I should see "Theme" in the modal

  Scenario: Different modals on the same page
```

```
Given I am a logged in user with the "admin" user
When I go to "/admin/users"

# First modal: Add user
When I press "Add User"
And wait for the modal to appear
Then I should see a modal with title "Add New User"
When I close the modal
And wait for the modal to disappear

# Second modal: Import users
When I press "Import"
And wait for the modal to appear
Then I should see a modal with title "Import Users"
When I close the modal
And wait for the modal to disappear

# Third modal: Delete confirmation
When I click "Delete" in the "john@example.com" row
And wait for the modal to appear
Then I should see "Confirm Delete" in the modal
When I click "Cancel" in the modal
And wait for the modal to disappear
```

How It Works:

`Then I should see the "login" modal` searches for a modal by its identifier. It looks for:

```
#login          ← element with id="login"
.login          ← element with class="login"
[data-modal="login"]  ← element with data-modal attribute
```

This is useful when your page can trigger different modals and you need to verify which one appeared. The title check (`modal with title "X"`) is another way to distinguish between modals when identifiers are not available.

The third scenario hides a timing rule worth making explicit: between modals, **always wait for the previous one to disappear before triggering the next**. `wait for the modal to appear` is satisfied by *any* visible modal, if the Add User modal is still fading out when you press Import, the wait matches the dying modal instantly, and the title assertion fails one step later pointing you at the wrong suspect. The disappear-wait between each pair is what keeps the three modals from contaminating each other's assertions. Field assertions inside the identified modal ("Email" and "Password" in the login modal) then confirm not just *which* modal opened but that it opened *complete*.

At scale: Treat modal identifiers the way Chapter 6 treats `data-test` attributes: a contract with the frontend team. A `data-modal="login"` attribute survives restyles, class renames, and framework migrations, the three things guaranteed to happen across a portfolio's lifetime, while `.login` classes come and go with every theme. Agree on the attribute convention once, document it where frontend developers will see it, and the same login-modal feature file ports between projects with nothing but the `LAUNCH_URL` changing.

Recipe 80 Testing Native Browser Dialogs

Problem: The legacy admin section still calls `confirm("Delete everything?")`, nobody has touched that code since 2019. Playwright's default is to dismiss any dialog it was not told about, so in your test the confirm silently returns `false`, the delete never runs, and the scenario fails with the least helpful evidence imaginable: no modal in the failure screenshot, nothing unusual in the DOM, because native dialogs live entirely outside it. Until you know that one default, debugging this costs an afternoon.

Solution:

Native browser dialogs require custom step definitions because they are not part of the DOM. They are handled through Playwright's `page.on('dialog')` event.

```
Feature: Native Browser Dialogs
  Scenario: Accept an alert dialog
    Given I am on homepage
    When I go to "/demo/alerts"
    And I press "Show Alert"
    # Native alert will be auto-accepted by Playwright
    Then I should see "Alert was closed"

  Scenario: Accept a confirm dialog
    Given I am on homepage
    When I go to "/demo/confirm"
    And I will accept the next browser dialog
    And I press "Delete Everything"
    Then I should see "Confirmed"

  Scenario: Dismiss a confirm dialog
    Given I am on homepage
    When I go to "/demo/confirm"
    And I will dismiss the next browser dialog
    And I press "Delete Everything"
    Then I should see "Cancelled"

  Scenario: Enter text in a prompt dialog
    Given I am on homepage
    When I go to "/demo/prompt"
    And I will enter "My Name" in the next browser dialog
    And I press "Ask Name"
    Then I should see "Hello, My Name"
```

Custom Step Definitions:

These steps are not built into webship-js, but you can add them to your project's step definitions:

```
// tests/step-definitions/custom-dialogs.js
const { When } = require('@cucumber/cucumber');

When('I will accept the next browser dialog', async function () {
  this.page.once('dialog', async (dialog) => {
    await dialog.accept();
  });
});

When('I will dismiss the next browser dialog', async function () {
  this.page.once('dialog', async (dialog) => {
    await dialog.dismiss();
  });
});

When('I will enter {string} in the next browser dialog', async function
(text) {
  this.page.once('dialog', async (dialog) => {
    await dialog.accept(text);
  });
});
```

How It Works:

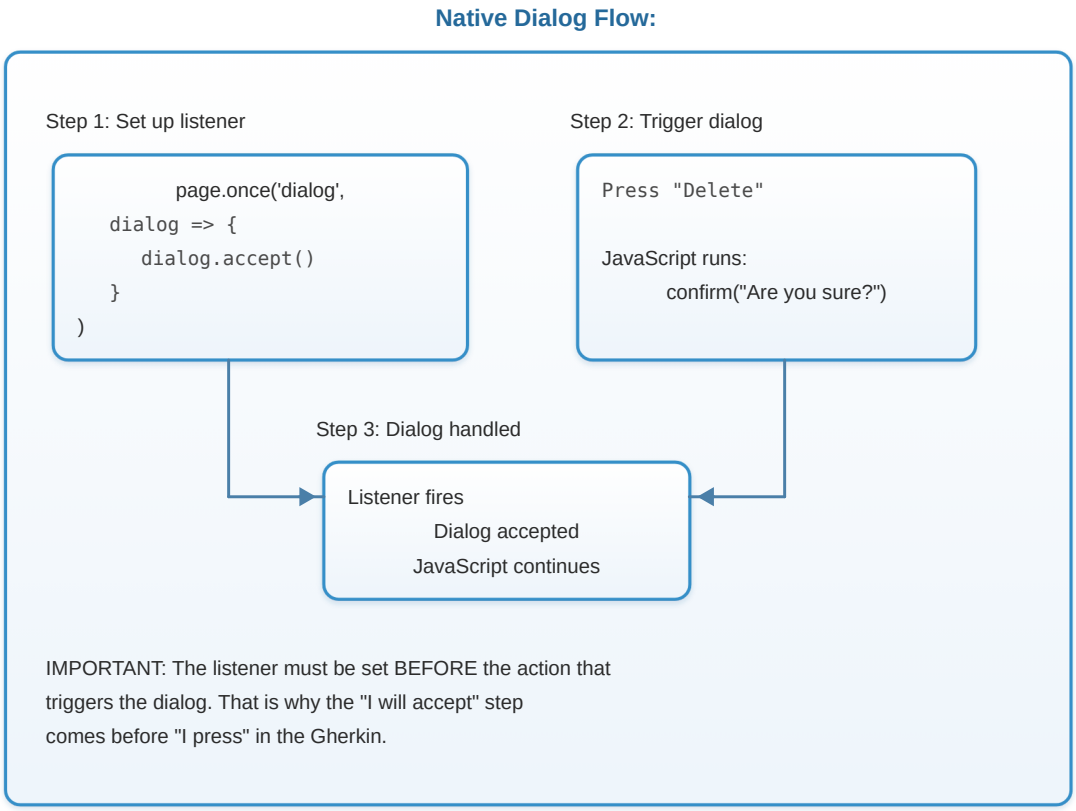


FIGURE 8.5 *Testing Native Browser Dialogs (Recipe 80)*

Key points about native dialogs:

- Playwright auto-dismisses unhandled `alert()` dialogs
- `confirm()` returns `true` (accepted) or `false` (dismissed)
- `prompt()` returns the entered text (accepted) or `null` (dismissed)
- You must set up the listener BEFORE the action that opens the dialog
- `page.once('dialog', ...)` handles exactly one dialog

That auto-dismiss default cuts both ways. It keeps a stray `alert()` from freezing your whole suite, native dialogs block all JavaScript on the page, so an unhandled one would otherwise hang every step after it. But it also means a stray dialog is *invisible*: the suite sails past it and you never learn it fired. The "I will…" steps use `page.once` rather than `page.on` deliberately, each step arms exactly one expectation for exactly one dialog, so a second unexpected dialog still surfaces as unusual behaviour instead of being quietly swallowed forever. When you write your own handler, log `dialog.message()` before accepting, the text of the dialog in your test output is often the first hard evidence of which code path fired it.

At scale: These custom dialog steps are precisely the kind of code that gets copy-pasted into forty project repos and then drifts, one project's version accepts, another's logs, a third has a typo'd phrasing. Put them in one shared step-definition package that every suite installs, and hold them to the webship-js house rule: every custom step documented with at least five example lines. The phrasing then becomes portfolio vocabulary, any engineer dropped into any project knows `And I will accept the next browser dialog` means the same thing everywhere.

Recipe 81 Testing Toast/Snackbar Notifications

Problem: The toast lives for five seconds. On your machine the assertion catches it with seconds to spare; on a loaded CI runner the suite arrives late and asserts on a message that has already faded out. So the save *worked* and the test *failed*, the flake that makes developers stop reading red builds. The opposite bug ships too: a toast that never dismisses, stacking up over the Save button until the user cannot click it. Both directions need pinning down.

Solution:

```
Feature: Toast Notifications
  As a user
  I want to see feedback notifications after actions
  So that I know my actions succeeded or failed

  Scenario: Success toast after saving
    Given I am a logged in user with the "admin" user
```

```
    When I go to "/settings"
    And I fill in "Site Name" with "My Website"
    And I press "Save"
    And wait for AJAX to finish
    Then I should see "Settings saved successfully"

  Scenario: Error toast for invalid action
    Given I am a logged in user with the "admin" user
    When I go to "/admin/users"
    And I click "Delete" in the "admin@example.com" row
    And wait for the modal to appear
    And I click "Delete" in the modal
    And wait for AJAX to finish
    Then I should see "Cannot delete the last admin user"

  Scenario: Toast auto-dismisses
    Given I am a logged in user with the "admin" user
    When I go to "/settings"
    And I press "Save"
    And wait for AJAX to finish
    Then I should see "Settings saved"
    # Wait for toast to auto-dismiss (e.g., 5 seconds)
    And wait 5s
    Then I should not see "Settings saved"

  Scenario: Multiple toast messages
    Given I am a logged in user with the "admin" user
    When I go to "/admin/import"
    And I press "Import All"
    And wait for AJAX to finish
    Then I should see "3 records imported"
    And I should see "1 record skipped"
```

How It Works:

Toast notifications are regular DOM elements, no special steps are needed. The standard `should see` and `should not see` assertions work perfectly. What makes the recipe reliable in webship-js 2.0 is how the timing machinery treats the toast's dismiss timer. That timer is an ordinary `setTimeout`, so it is counted in `window.__webshipPendingTimers`, and the automatic settle after `I press "Save"` has a 1500ms budget, which elapses long before a 5-second timer fires. The settle returns, your `should see` runs while the toast is alive, and you are not left racing it. Then, in the auto-dismiss scenario, `wait 5s` holds for its *full* budget precisely because that pending timer keeps the page from settling, landing your `should not see` after the dismissal. The same mechanism serves both assertions. The key is understanding the toast lifecycle:

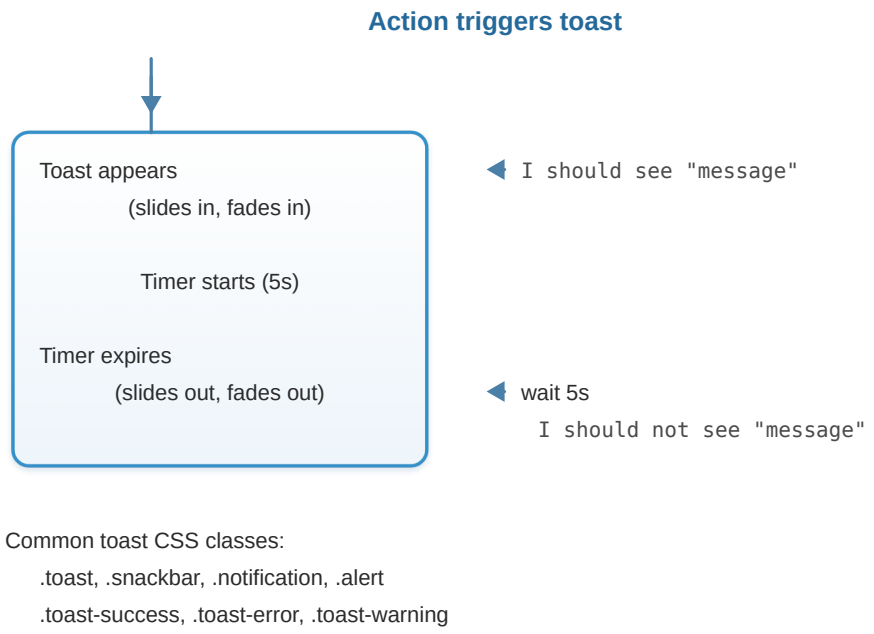


FIGURE 8.6 *Testing Toast/Snackbar Notifications (Recipe 81)*

When to test auto-dismiss:

Testing that a toast auto-dismisses is only necessary when the auto-dismiss duration is part of a business requirement. For most tests, verifying the toast appears is sufficient. Only add the timed wait + not-see check when:

- The auto-dismiss duration is a specific requirement
- A bug has been reported where toasts stay on screen permanently
- The toast blocks interaction with elements below it

At scale: Toast copy is one of the easiest things to standardize across a portfolio and one of the most valuable. When every product says “Settings saved successfully”, not “Saved!”, “Done”, and “Your changes have been applied” in three different apps, the same assertion lines port between suites unchanged, and the HTML report reads like a checklist a delivery manager can actually follow: action taken, confirmation shown, by name. Those message assertions are quiet release evidence; treat the copy like an API and put changes to it through review.

Recipe 82 Testing Cookie Consent and Banner Modals

Problem: Marketing rolls a consent manager out to all your sites on a Thursday afternoon, and by Friday morning every nightly suite is red. Here is why it hits tests harder than users: each scenario starts a fresh browser context with no cookies, so to the consent manager *every test is a first visit*, the banner appears every time, on every page, sitting on top of whatever the test wanted to click. A returning user sees it once; your suite sees it four hundred times a night until someone deals with it properly.

Solution:

```
Feature: Cookie Consent Banner
  Scenario: Accept cookies and continue browsing
    Given I am on homepage
    Then I should see "We use cookies"
    When I press "Accept All"
    Then I should not see "We use cookies"

    # Now continue with normal testing
    When I go to "/about"
    Then I should see "About Us"
    # Cookie banner should not reappear
    And I should not see "We use cookies"

  Scenario: Reject optional cookies
    Given I am on homepage
    Then I should see "We use cookies"
    When I press "Reject Optional"
    Then I should not see "We use cookies"
    # Essential functionality should still work
    When I go to "/products"
    Then I should see "Products"

  Scenario: Customize cookie preferences via modal
    Given I am on homepage
    Then I should see "We use cookies"
    When I press "Customize"
    And wait for the modal to appear
    Then I should see "Cookie Preferences" in the modal
    And I uncheck "Marketing Cookies"
    And I uncheck "Analytics Cookies"
    When I click "Save Preferences" in the modal
    And wait for the modal to disappear
    Then I should not see "We use cookies"
```

How It Works:

Cookie banners are a special case of modal/notification testing. They typically appear on first visit and must be handled before other interactions. You have two strategies:

```
Strategy 1: Handle in the test
(Explicit, shown in recipe above)

Feature file:
  Given I am on homepage
  When I press "Accept All"
banner
  ...continue testing...
this.context.addCookies([{
```

```
Strategy 2: Handle in a hook
(Automatic, for every test)

Before hook:
  Before(async function() {
    // Set cookie to skip

    await

     name: 'cookie_consent',
     value: 'accepted',
     domain: 'localhost',
     path: '/'
    }]);
  });
```

Use Strategy 1 when you are specifically testing the cookie consent flow. Use Strategy 2 when the cookie banner is just in the way of other tests.

Strategy 2 works because consent managers decide whether to show the banner by reading a cookie before the page renders, `context.addCookies()` in a Before hook plants that cookie ahead of the first navigation, so the banner never mounts at all. Strategy 1, meanwhile, can assert more than the banner's disappearance: after pressing "Accept All", webship-js's cookie steps let you pin the actual mechanism with `Then a cookie with the name "cookie_consent" should exist`. That upgrades the scenario from "the banner went away" to "the *choice was recorded*", which is the part a privacy audit cares about. The first scenario's final check, navigating to another page and asserting the banner stays gone, tests the same persistence from the user's side.

At scale: When the consent manager lands on twenty sites at once, the Before hook belongs in shared support code, with the cookie name and domain derived from configuration rather than hardcoded, note the `domain: 'localhost'` in the example, which must follow `LAUNCH_URL` per environment or the hook silently plants a cookie no site reads. Then keep exactly one tagged

scenario per site that runs Strategy 1 against the real banner, so the consent flow itself stays tested while the other four hundred scenarios skip it. One shared hook, one real test per site: the Thursday rollout becomes a one-line change instead of a portfolio-wide fire drill.

Recipe 83 Testing Drawer and Sidebar Panels

Problem: On phones your navigation is a drawer; on desktop the same links live in a top bar. The drawer slides in over 300ms, and a test that asserts mid-slide sees half a menu, or clicks a link that is still moving and watches the click land on yesterday's coordinates. Drawer tests are also where viewport assumptions hide: the scenario that passes on a desktop-sized CI runner may be testing a menu that does not even *render* as a drawer at that width.

Solution:

```
Feature: Drawer Panel
  As a user on a mobile-responsive site
  I want to access the navigation drawer
  So that I can navigate the application

  Scenario: Open and close navigation drawer
    Given I am on homepage
    When I press "Menu"
    And wait 1s
    Then I should see "Navigation"
    And I should see "Home"
    And I should see "Products"
    And I should see "Contact"

    When I click "Products"
    And wait for AJAX to finish
    Then I should see "Products"

  Scenario: Settings drawer panel
    Given I am a logged in user with the "admin" user
    When I go to "/dashboard"
    And I press "Settings"
```

```
    And wait 1s
    Then I should see "Dashboard Settings"
    And I should see "Theme"
    And I should see "Layout"
    When I select "Dark" from "Theme"
    And I press "Apply"
    And wait for AJAX to finish
    Then I should see "Settings applied"

  Scenario: Detail drawer in list view
    Given I am a logged in user with the "admin" user
    When I go to "/admin/orders"
    And I click "View" in the "ORD-001" row
    And wait 1s
    Then I should see "Order Details"
    And I should see "ORD-001"
    And I should see "Status: Processing"
```

How It Works:

Drawers are technically similar to modals but with different visual behavior. They slide in from the side rather than appearing in the center. From a testing perspective:

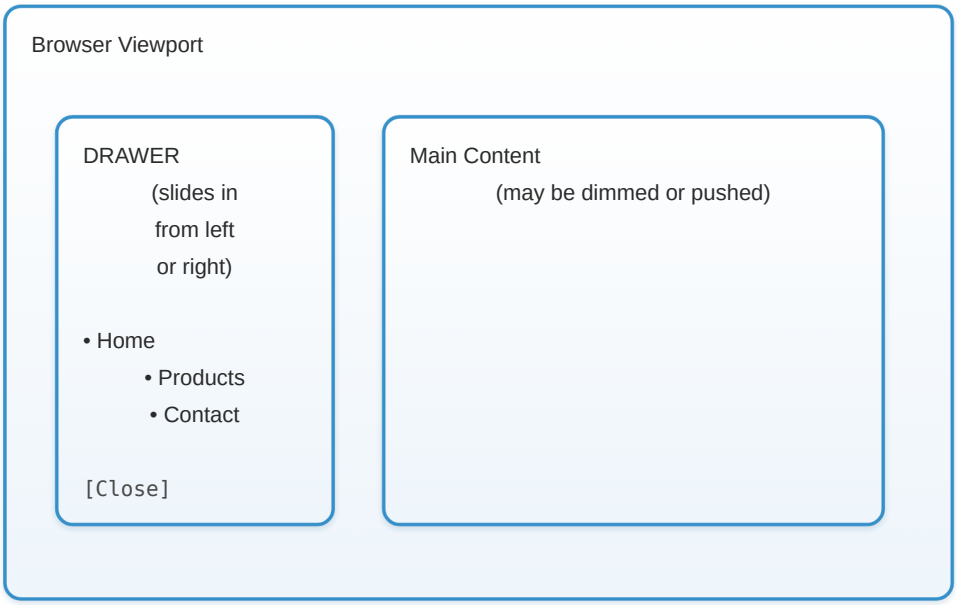


FIGURE 8.7 *Testing Drawer and Sidebar Panels (Recipe 83)*

If the drawer uses `role="dialog"` or `.modal` classes, webship-js modal steps work directly. If it uses custom classes (like `.drawer`, `.sidebar`, `.off-canvas`), use the standard `should see` / `should not see` text assertions instead.

The `wait 1s` after the trigger handles the slide-in animation, and it is worth understanding why an explicit wait is legitimate here when this book keeps telling you to avoid them. A CSS slide transition is compositor work: no AJAX fires, no `setTimeout` is scheduled, and the DOM mutates exactly once, the class flip that starts the slide. All three of `smartSettle`'s signals go quiet while the panel is still visibly moving, so the automatic post-step settle can return mid-animation. The transition's duration is part of the UI, which puts this squarely in Recipe 69's "time is the behaviour" category; most slide transitions run 300-500ms, so a 1-second budget covers them with

margin. If your drawer has a stable selector, the web-first form is tighter still: `Then ".drawer" should be visible` polls every 100ms for up to 5 seconds and returns the instant the panel reports visible.

At scale: Decide, once, for the whole portfolio, which viewport your shared suites run at. webship-js's selectors registry defines named breakpoints from `xs` at 375x667 up to `xxxl` at 1920x1080, defaulting to `xl` at 1200x900 (override via `WEBSHIP_SELECTORS_BREAKPOINTS`), and a drawer that exists at `xs` may be a top bar at `xl`. The pattern that holds up across many projects: a desktop suite as the default, plus a small mobile-tagged suite pinned to one phone-sized breakpoint for the drawer, the hamburger, and the touch flows. Twenty projects each improvising their own viewport is how you get twenty subtly different definitions of "the menu works."

Recipe 84 Testing Nested Modals

Problem: The email-settings modal has an "Advanced Settings" button that opens a second modal on top of the first. The classic bug ships in the seam between them: Escape or Save closes *both* layers at once, eating the unsaved changes in the outer modal, or the inner modal closes and leaves the outer one's backdrop frozen, locking the whole page. Nested modals are a UX smell, but enterprise admin screens grow them anyway, and somebody has to prove the layers open and close independently.

Solution:

```
Feature: Nested Modals
  Scenario: Modal opens a second modal
    Given I am a logged in user with the "admin" user
    When I go to "/admin/settings"
    And I press "Configure Email"
    And wait for the modal to appear
    Then I should see "Email Configuration" in the modal

    # Open nested modal
    When I click "Advanced Settings" in the modal
    And wait 1s
    Then I should see "Advanced Email Settings" in the modal
    And I should see "SMTP Host" in the modal
    When I fill in "SMTP Host" with "smtp.example.com"
    And I click "Save" in the modal
    And wait for AJAX to finish

    # Back to the first modal
    Then I should see "Email Configuration" in the modal
    When I click "Close" in the modal
    And wait for the modal to disappear
    Then I should not see a modal dialog
```

How It Works:

Nested modals are tricky because:

1. Multiple modal elements may be visible simultaneously
2. The “top” modal should capture all interactions
3. Closing the inner modal should return to the outer modal

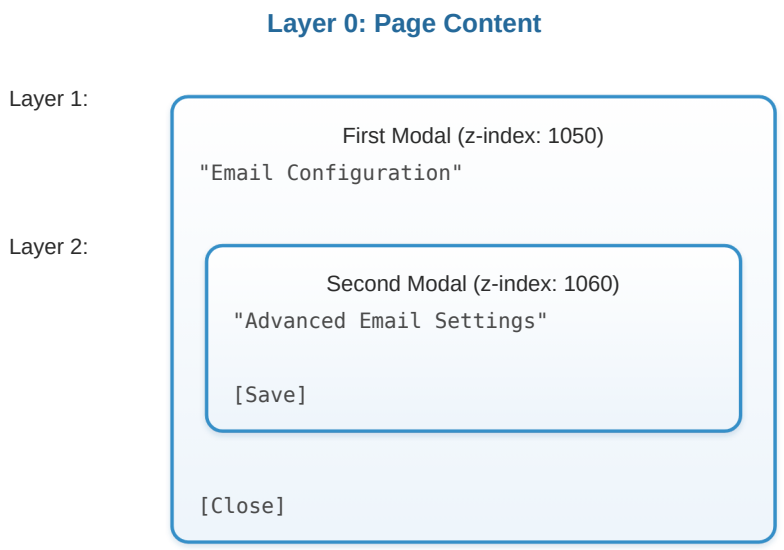


FIGURE 8.8 *Testing Nested Modals (Recipe 84)*

webship-js `findVisibleModal()` returns the **first** visible modal it finds. When nested modals are present, browser frameworks typically give the topmost modal the highest z-index, but "highest z-index" and "first in the DOM" are not the same element, so the safest scenarios disambiguate by *content*: the `click "X" in the modal` step interacts with whichever visible modal contains the matching button, which is why the scenario clicks button captions that exist in only one layer ("Save" lives in the inner modal, "Close" in the outer).

The load-bearing assertion comes after the inner save: `Then I should see "Email Configuration" in the modal`. That line proves the **outer modal survived** the inner one's close, it is the direct test for the both-layers-collapse bug, and without it the scenario would pass right through that failure. The final disappear-wait then confirms the reverse: closing the outer modal really does end the stack, with no orphaned backdrop left blocking the page.

Best Practice: Avoid nested modals when possible. They create a confusing user experience and are harder to test. If your application uses them, test the complete flow: open outer → open inner → close inner → verify outer → close outer. And if a designer asks whether the sub-modal could just be a second *page* instead, agree enthusiastically.

Recipe 85 Testing Modal with Tab Navigation

Problem: A tabbed modal is three forms wearing one title, and the bugs live in the seams between the tabs. Switching from General to Security silently wipes the first name the user just typed; the Notifications tab lazy-loads and a fast click lands before its fields exist; Save only submits the tab you happen to be looking at. Users hit these daily in profile and settings dialogs, and a test that only visits one tab certifies a third of the modal.

Solution:

```
Feature: Tabbed Modal
  Scenario: Navigate tabs inside a modal
    Given I am a logged in user with the "admin" user
    When I go to "/profile"
    And I press "Edit Profile"
    And wait for the modal to appear
    Then I should see "Edit Profile" in the modal

    # First tab (default)
    Then I should see "First Name" in the modal
    And I should see "Last Name" in the modal

    # Switch to second tab
    When I click "Security" in the modal
    And wait 1s
    Then I should see "Current Password" in the modal
    And I should see "New Password" in the modal

    # Switch to third tab
    When I click "Notifications" in the modal
    And wait 1s
    Then I should see "Email Notifications" in the modal
    And I should see "SMS Notifications" in the modal

    # Go back to first tab and make changes
    When I click "General" in the modal
    And wait 1s
    And I fill in "First Name" with "Rajab"
    And I click "Save" in the modal
    And wait for AJAX to finish
    Then I should not see a modal dialog
    And I should see "Profile updated"
```

How It Works:

The tab clicks are scoped (`click "Security" in the modal`), so a page that also has a "Security" link in its navigation cannot steal the click, the search never leaves the visible modal. Each `wait 1s` covers the tab-pane transition, which is the same compositor-only animation as Recipe 83's drawer: a single class flip and a CSS fade that none of `smartSettle`'s counters can see, so a small explicit budget is the honest tool.

Two details make this scenario stronger than a simple tab walk. First, each tab switch is followed by assertions on fields that exist *only* in that pane, "Current Password" can only mean the Security tab rendered, not merely highlighted its tab button. Second, inactive panes typically keep their fields in the DOM but hidden, and `fill in` targets the visible match, which is why the scenario navigates back to General *before* typing the first name. Fill while the wrong tab is active and the step either finds nothing or, worse, a hidden duplicate.

The ending carries the real verdict: filling on tab one, after visiting tabs two and three, then saving and seeing "Profile updated" proves edits survive tab switches, the exact thing the wipe-on-switch bug breaks. If your team has been bitten by that bug before, extend the pattern: type on General, visit Security, come back, and assert the value is still in the field with `Then I should see "Rajab" value in the "First Name" input element` before saving.

Recipe 86 Testing Image Lightbox

Problem: Galleries break at their edges. The lightbox opens fine on every image, then "Next" on the last photo shows an empty pane, the captions drift one image out of sync with the photos, or the X button stops closing after the library update nobody tested. These are the bugs your client notices during the demo of their own portfolio site, because the gallery is the one page they always show off.

Solution:

```
Feature: Image Lightbox
  Scenario: Open lightbox from gallery
    Given I am on homepage
    When I go to "/gallery"
    And wait for AJAX to finish
    Then I should see "Photo Gallery"

    # Click a thumbnail to open lightbox
    When I click "Beach Sunset"
    And wait 1s
    Then I should see a modal dialog
    And I should see "Beach Sunset" in the modal

    # Close lightbox
    When I close the modal
    And wait for the modal to disappear
    Then I should not see a modal dialog

  Scenario: Navigate lightbox gallery
    Given I am on homepage
    When I go to "/gallery"
    And I click "Beach Sunset"
    And wait 1s
    Then I should see a modal dialog
    And I should see "Beach Sunset" in the modal

    # Navigate to next image
    When I click "Next" in the modal
    And wait 1s
    Then I should see "Mountain View" in the modal

    # Navigate to previous image
    When I click "Previous" in the modal
    And wait 1s
    Then I should see "Beach Sunset" in the modal
```

How It Works:

Most lightbox libraries render as exactly the kind of element webship-js's modal selector already matches, `.popup`, `.modal`, or a `[role="dialog"]` overlay, so the standard modal steps work without any custom code: the appear/disappear waits, the scoped clicks for "Next" and "Previous", and `close the modal` for the X button.

The captions are doing the clever work. You cannot cheaply assert "the right *image* is showing", that would mean pixel comparison, but a caption rendered alongside its image carries the image's identity as text, and `I should see "Mountain View" in the modal` becomes a precise statement about which photo is on screen. This is also why the second scenario goes forward *and then back*: "Next" landing on Mountain View proves navigation advances, but "Previous" returning to Beach Sunset is what catches the off-by-one index bugs and caption drift that one-direction clicking sails past. The `wait 1s` after each navigation covers the crossfade between images, compositor animation again, invisible to the settle counters.

Pro Tip: If the gallery is content-managed, the captions in this scenario are seed data. Keep the test gallery's images and captions in your fixtures rather than relying on whatever the client uploaded last week, or the suite breaks every time marketing refreshes the photos.

Recipe 87 Testing Tooltips

Problem: Icon-only toolbars put the entire explanation of a button into its tooltip, and then a redesign swaps the tooltip library, the new one ignores the data attributes the old one read, and every hint on the site silently disappears. No error, no broken layout, just an interface that stopped explaining itself. Nobody hovers in automated tests by default, so the regression rides through three releases until a confused-user ticket finally names it.

Solution:

Tooltips require hover interaction. webship-js does not have a built-in hover step, but you can create one:

```
Feature: Tooltips
  Scenario: Tooltip shows on hover
    Given I am on homepage
    When I go to "/dashboard"
    And I hover over "Help Icon"
    And wait 1s
    Then I should see "Click here for help"

  Scenario: Tooltip on disabled button
    Given I am on homepage
    When I go to "/settings"
    And I hover over "Delete Account"
    And wait 1s
    Then I should see "Contact admin to delete your account"
```

Custom Step Definition:

```
// tests/step-definitions/custom-hover.js
const { When } = require('@cucumber/cucumber');

When('I hover over {string}', async function (text) {
  const element = this.page.getByText(text).first();
  await element.hover();
});

When('I hover over the {string} element', async function (selector) {
  const builtSelector = buildSelector(selector);
  await this.page.locator(builtSelector).first().hover();
});
```

How It Works:

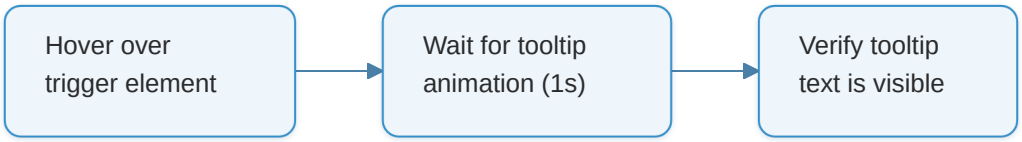


FIGURE 8.9 *Testing Tooltips (Recipe 87)*

TYPE	HOW IT WORKS
CSS-only tooltip	Uses :hover pseudo-class
title attribute	Browser renders native tooltip
Bootstrap tooltip	JS inserts
aria-describedby	Shows linked element on focus
Custom JS tooltip	JS positions element on hover

Note: Native `title` attribute tooltips cannot be reliably tested with Playwright because browsers render them outside the DOM. If your application uses `title` attributes, consider switching to JavaScript-based tooltips that are part of the DOM and can be asserted against. You *can* still pin the attribute itself, `Then "#help-icon" should have attribute "title" with value "Click here for help"` verifies the text is wired up, even though no browser-drawn bubble will ever appear in a screenshot.

The custom step itself is small because Playwright's `element.hover()` does the hard part: it scrolls the element into view, moves the virtual pointer to it, and fires the real `mouseenter`/`mouseover` chain, so CSS `:hover` rules and JavaScript hover listeners both trigger exactly as they would for a human hand on a mouse. The `wait 1s` that follows covers the show-delay most tooltip libraries add (typically 200-500ms, deliberately, so tooltips do not

flicker as the pointer crosses the screen). The disabled-button scenario earns its place: explaining *why* an action is unavailable is the tooltip pattern with the most user-experience weight, and the one support tickets quote back at you.

At scale: The hover steps belong in your shared custom-step package, not copied into each project, same rule as the dialog steps in Recipe 80, same five-example documentation standard. And remember that hover does not exist on touch screens: if your portfolio includes mobile-web suites, any information that lives only in a tooltip is information a phone user never gets. That is an accessibility and design finding, not just a testing inconvenience, raise it once at the design-system level and every project inherits the fix.

Recipe 88 Testing Popovers

Problem: Admin tables tuck their row actions into click-triggered popovers, the little “Actions” menu with Edit, Duplicate, Archive, Delete. The recurring bug: the popover opens but click-away never closes it, so it hangs there overlapping the next row’s menu, and the second action the user takes lands on the wrong record. The open half of a popover gets tested because it has to be opened to be used; the *close* half is the part that quietly rots.

Solution:

```
Feature: Popovers
  Scenario: Popover shows on click
    Given I am on homepage
    When I go to "/contacts"
    And I click "John Doe"
    And wait 1s
    Then I should see "Contact Details"
    And I should see "john@example.com"
    And I should see "555-0123"

    # Click elsewhere to close popover
    When I press "Contacts"
    And wait 1s
    Then I should not see "Contact Details"

  Scenario: Action popover menu
    Given I am a logged in user with the "admin" user
    When I go to "/admin/posts"
    And I press "Actions"
    And wait 1s
    Then I should see "Edit"
    And I should see "Duplicate"
    And I should see "Archive"
    And I should see "Delete"
    When I click "Edit"
    And wait for AJAX to finish
    Then I should see "Edit Post"
```

How It Works:

Popovers are similar to tooltips but triggered by click instead of hover and contain richer content. From a testing perspective, they are just DOM elements that appear and disappear:

FEATURE	TOOLTIP	POPOVER	MODAL
Trigger	Hover	Click	Click
Content	Text only	Rich HTML	Full forms
Backdrop	No	No	Yes (usually)
Blocks page	No	No	Yes
Position	Near elem	Near elem	Center screen
Close trigger	Mouse out	Click away	Close button/ESC
Testing approach	hover+see	click+see	modal steps

The “Click away” row in that table is the one the first scenario exercises deliberately. Popovers have no backdrop and no close button, dismissal *is* clicking somewhere else, which the scenario does with `When I press "Contacts"`, and the `Then I should not see "Contact Details"` that follows is the actual test of the close behaviour users depend on. Skip that negative assertion and the stuck-popover bug from the problem statement passes clean. The action-menu scenario then covers the other contract: every item in the menu is present before you commit to one, so a permissions change that drops “Delete” from the menu fails loudly instead of silently shrinking the UI.

The Complete Modal Testing Pattern

Here is a comprehensive example that combines many modal patterns into a realistic admin

workflow:

```
Feature: Product Management with Modals
  As an admin managing an e-commerce store
  I want to use modals for quick product operations
  So that I can manage products without navigating away from the list

  Background:
    Given I am a logged in user with the "admin" user
    And I go to "/admin/products"
    And wait for AJAX to finish

  Scenario: Complete modal CRUD workflow
    # CREATE via modal
    When I press "Add Product"
    And wait for the modal to appear
    Then I should see a modal with title "Add Product"
    When I fill in "Product Name" with "Wireless Mouse"
    And I fill in "Price" with "29.99"
    And I fill in "SKU" with "WM-001"
    And I select "Electronics" from "Category"
    And I fill in "Description" with "Ergonomic wireless mouse"
    And I click "Save" in the modal
    And wait for AJAX to finish
    Then I should not see a modal dialog
    And I should see "Product created"
    And I should see "Wireless Mouse"

    # READ/VIEW via modal
    When I click "View" in the "Wireless Mouse" row
    And wait for the modal to appear
    Then I should see "Wireless Mouse" in the modal
    And I should see "$29.99" in the modal
    And I should see "WM-001" in the modal
    When I close the modal
    And wait for the modal to disappear

    # UPDATE via modal
    When I click "Edit" in the "Wireless Mouse" row
    And wait for the modal to appear
```

```
Then I should see a modal with title "Edit Product"
When I fill in "Price" with "24.99"
And I click "Save Changes" in the modal
And wait for AJAX to finish
Then I should not see a modal dialog
And I should see "Product updated"

# DELETE via confirmation modal
When I click "Delete" in the "Wireless Mouse" row
And wait for the modal to appear
Then I should see "Are you sure you want to delete" in the modal
And I should see "Wireless Mouse" in the modal
When I click "Yes, Delete" in the modal
And wait for AJAX to finish
Then I should not see a modal dialog
And I should see "Product deleted"
And I should not see "Wireless Mouse"
```

A fair question: does one scenario doing create, view, update, *and* delete violate the one-behaviour-per-scenario rule? Here, no, and the reason is worth internalizing. Each stage consumes the previous stage's data: you can only view, edit, and delete the Wireless Mouse because the first block created it. Splitting this into four scenarios would force each one to re-create its own product (quadrupling runtime) or to depend on leftover state from a previous scenario (the cardinal sin of test design). The behaviour under test is the full life of one record through the modal UI, and the scenario ends having cleaned up after itself, the delete is both the last test and the teardown.

At scale: This workflow is the same on every admin UI in your portfolio, only the nouns change. Keep it as the template scenario for "entity managed through modals": copy it, swap the field names and button captions, and a new project has its core CRUD coverage in an afternoon. Because it creates and deletes its own record, it is safe to run against any environment, repeatedly, in parallel with other suites, which is exactly the property you need when forty pipelines share a staging farm.

Modal Testing Quick Reference

WHAT YOU WANT	STEP TO USE
Wait for modal to open	And wait for the modal to
	appear
Wait for modal to close	And wait for the modal to
	disappear
Verify any modal is visible	Then I should see a modal
	dialog
Verify no modal is visible	Then I should not see a modal
	dialog
Verify modal title	Then I should see a modal with
	title "X"
Verify text in modal	Then I should see "X" in the
	modal
Verify specific modal by ID	Then I should see the "X"
	modal
Click button in modal	When I click "X" in the modal
Close modal (close button/ESC)	When I close the modal
Dismiss modal (same as close)	When I dismiss the modal

WHAT YOU WANT	STEP TO USE
Fill form fields in modal	Standard form steps (they work
	on visible elements)
Handle native alert	Custom step + page.on('dialog')
Handle native confirm	Custom step + dialog.accept()
	or dialog.dismiss()
Handle native prompt	Custom step + dialog.accept
	(text)

Chapter Summary

- Two types of dialogs: **native browser** (alert/confirm/prompt) and **custom modals** (DOM elements)
- webship-js has built-in steps for custom modals; native dialogs need custom step definitions
- Always **wait for the modal to appear** before interacting with it
- Always **wait for the modal to disappear** after closing it
- Modal detection covers: `.modal`, `.modal.show`, `[role="dialog"]`, `.dialog`, `.popup`
- `click "X" in the modal` works for both buttons and links inside modals
- `close the modal` tries the close button first, then falls back to Escape key
- Form steps work inside modals because they target visible DOM elements
- Toast/snackbar notifications are just regular DOM elements, use `should see` / `should not see`
- Cookie consent banners can be handled in tests or bypassed with cookies in hooks
- Drawers and sidebars follow the same open/interact/close pattern
- Avoid nested modals when possible, they complicate both UX and testing
- Native `title` attribute tooltips cannot be reliably tested, use JS-based tooltips

Next: Table and List Testing recipes.

Chapter 9: Table and List Testing

Tables are the backbone of data-driven web applications. Admin panels, dashboards, reports, user management screens, order lists, inventory systems, all of them present data in tabular form. If your application has a backend, it has tables.

Testing tables introduces a unique challenge: you are not just checking that text exists on the page, you need to verify that the right data appears in the right row. The product name “Laptop” should appear in the same row as the price “$999” and the status “In Stock.” A simple `I should see "Laptop"` tells you the text is somewhere on the page but not whether it is in the correct context.

This chapter gives you recipes for every table interaction pattern: reading data, clicking actions within rows, sorting, filtering, pagination, bulk operations, and more.

Understanding Table Structure

Before we dive into recipes, let us understand the HTML structure that webship-js targets:

HTML Table Structure:

```
<table>
  <thead>
    <tr>                                                  ◀ Header Row
      <th>Name</th>
      <th>Email</th>
      <th>Role</th>
      <th>Actions</th>
    </tr>
  </thead>
  <tbody>
    <tr>                                                  ◀ Data Row 1
      <td>John Smith</td>
      <td>john@example.com</td>
      <td>Admin</td>
      <td><a>Edit</a> <button>Delete</button></td>
    </tr>
    <tr>                                                  ◀ Data Row 2
      <td>Jane Doe</td>
      <td>jane@example.com</td>
      <td>Editor</td>
      <td><a>Edit</a> <button>Delete</button></td>
    </tr>
  </tbody>
</table>
```

FIGURE 9.1 *Understanding Table Structure*

webship-js row-based steps work by:

1. Finding all `<tr>` elements on the page
2. Filtering to the row that contains the **row identifier** text
3. Performing the action within that specific row

This means you can interact with any cell in a row by identifying the row with any unique text it contains (name, email, ID, etc.).

Recipe 89 Verifying Table Content

Problem: Your admin user list shows the right names and the right roles, until the day it does not. A permissions refactor swaps two columns in the API response, and suddenly every user displays someone else's role. Your old test passed anyway, because `I should see "Admin"` only proves the word "Admin" exists *somewhere* on the page. What you actually need to prove is that "Admin" sits in the same row as john@example.com, and that is a different assertion entirely.

Solution:

```
Feature: Table Content Verification
  As an admin
  I want to see user data in a table
  So that I can manage the system

  Scenario: Table shows expected data
    Given I am a logged in user with the "admin" user
    When I go to "/admin/users"
    And wait for AJAX to finish
    Then I should see "John Smith"
    And I should see "john@example.com"
    And I should see "Jane Doe"
    And I should see "jane@example.com"

  Scenario: Verify data in specific rows
    Given I am a logged in user with the "admin" user
    When I go to "/admin/users"
    And wait for AJAX to finish
    Then I should see "Admin" in the "john@example.com" row
    And I should see "Editor" in the "jane@example.com" row
    And I should see "Active" in the "john@example.com" row
    And I should see "Active" in the "jane@example.com" row

  Scenario: Verify data NOT in a row
    Given I am a logged in user with the "admin" user
    When I go to "/admin/users"
    And wait for AJAX to finish
    Then I should not see "Blocked" in the "john@example.com" row
    And I should not see "Admin" in the "jane@example.com" row
```

How It Works:

`Then I should see "Admin" in the "john@example.com" row` does the following:

```
// What webship-js does internally
const rows = Array.from(document.querySelectorAll('tr'));
const found = rows
  .filter(r => r.textContent.includes('john@example.com'))  // Find the
row
  .some(r => r.textContent.includes('Admin'));              // Check for
text
```

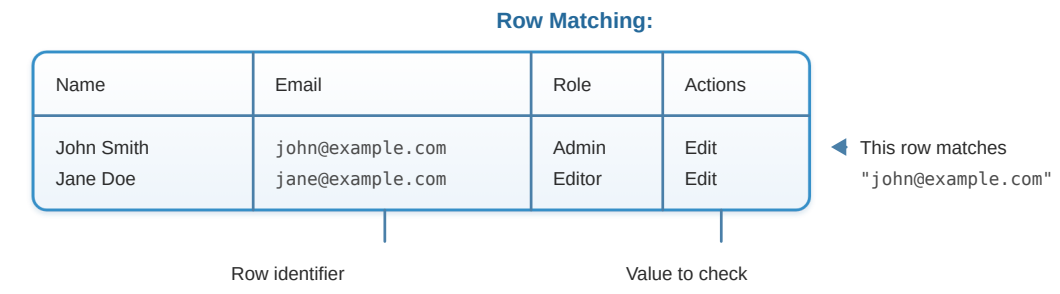


FIGURE 9.2 *Verifying Table Content (Recipe 89)*

The two-text approach is what makes this assertion meaningful. The row identifier narrows the search space to one `<tr>`; the value check then runs only inside that row's text content. If "Admin" appears in Jane's row but not John's, the step fails, exactly the column-swap bug a page-wide `should see` would have missed. Note also the `wait for AJAX to finish` before the assertions: admin tables almost always load their rows through a fetch after the page shell renders, and webship-js tracks those in-flight requests through its `window.__webshipAjaxCount` wrapper, so the assertion only runs once the data has actually landed in the DOM.

Choosing good row identifiers:

The row identifier should be a value that uniquely identifies a single row in the table. Good choices:

- Email addresses (unique by nature)
- IDs or reference numbers (`ORD-12345`)
- Usernames
- Full names (if unique in the test data)

Bad choices:

- Status values ("Active", multiple rows might have this)
- Common text ("Edit", every row has an Edit button)

At scale: When you run this pattern across many projects, the failure mode is not the step, it is the test data. One project seeds john@example.com, another seeds admin@client-a.test, and your scenarios stop being portable. Standardize the seed fixtures: the same set of test users with the same emails and roles in every project, loaded by the same seed script. Then the same `in the "..." row` scenarios can be copied between suites verbatim, and a new QA engineer can read any project's table tests without asking who "Bob Wilson" is supposed to be.

Recipe 90 Clicking Actions in Table Rows

Problem: Every row in your user table has an Edit link and a Delete button, fifty rows, fifty identical "Edit" labels. A plain `When I click "Edit"` grabs the first one it finds, which is fine until a new user is seeded above your target row and the test quietly starts editing the wrong account. The day someone notices, the test has been "passing" for three weeks while validating nothing. You need the click scoped to a specific row, not to whichever row happens to be first.

Solution:

```
Feature: Table Row Actions
  Background:
    Given I am a logged in user with the "admin" user
    When I go to "/admin/users"
    And wait for AJAX to finish

  Scenario: Edit a user from the table
    When I click "Edit" in the "john@example.com" row
    And wait for AJAX to finish
    Then I should see "Edit User"
    And I should see "John Smith"

  Scenario: Delete a user from the table
    When I click "Delete" in the "jane@example.com" row
    And wait for the modal to appear
    Then I should see "Are you sure?" in the modal
    When I click "Confirm" in the modal
    And wait for AJAX to finish
    Then I should not see "jane@example.com"

  Scenario: View details from the table
    When I click "View" in the "john@example.com" row
    And wait for AJAX to finish
    Then I should see "User Details"
    And I should see "John Smith"
    And I should see "Admin"

  Scenario: Toggle status from the table
    Then I should see "Active" in the "john@example.com" row
    When I click "Deactivate" in the "john@example.com" row
    And wait for AJAX to finish
    Then I should see "Inactive" in the "john@example.com" row
```

How It Works:

`When I click "Edit" in the "john@example.com" row` does the following:

1. Finds the `<tr>` that contains "john@example.com"
2. Within that row, searches for `<a>`, `<button>`, `[role="button"]`, `.btn`, or `<input type="submit">` elements
3. Filters to the one with text matching "Edit"
4. Clicks it

Search Strategy:

```
<tr>
   <td>John Smith</td>
   <td>john@example.com</td>          ◀ Row found by this text
   <td>Admin</td>
   <td>
      <a href="/edit/1">Edit</a>      ◀ Clickable match!
      <button>Delete</button>         ◀ Not this one
   </td>
</tr>
```

`Priority: a > button > [role="button"] > .btn > input[type="submit"]`
If no clickable element matches, falls back to getByText exact match.

FIGURE 9.3 *Clicking Actions in Table Rows (Recipe 90)*

The priority order matters more than it looks. Action cells often contain both a visible "Edit" link and a hidden form button with similar text; searching anchors first means the step clicks what a human would click. And because the search is scoped to the matched `<tr>`, the other forty-nine "Edit" links on the page are never candidates, the wrong-row bug from the Problem statement becomes structurally impossible.

After the click, the AfterStep hook runs `smartSettle(page, 1500)` automatically: it waits for in-flight fetch/XHR calls to drain and the DOM to go quiet before your next step runs. That is why "click Delete, confirm in modal, assert row gone" works without any sleep between steps.

At scale: Across a portfolio of admin UIs, the action labels drift, one project says “Edit”, another “Modify”, a third uses an icon-only button with an aria-label. Resist writing per-project step variants. Instead, push back on the products: a shared convention (“row actions are anchors or buttons with visible text or aria-labels from an agreed vocabulary”) costs the developers nothing and lets one set of scenarios cover every project. This is a conversation a QA lead should have once, not a workaround each suite carries forever.

Recipe 91 Testing Table with CRUD Operations

Problem: Create works. Edit works. Delete works. You have tested each one, in isolation, against pre-seeded data. Then a release ships where creating a product succeeds but the new row never appears in the list, because the list query caches aggressively. No isolated test catches that; only the full lifecycle does: create it, see it in the table, change it, see the change, delete it, see it gone. This is the test that fails when the pieces stop talking to each other.

Solution:

```
Feature: Table CRUD Operations
  As an admin managing products
  I want to create, view, edit, and delete products
  So that I can maintain the product catalog

  Background:
    Given I am a logged in user with the "admin" user
    And I go to "/admin/products"
    And wait for AJAX to finish

  Scenario: Create a new product and see it in the table
    When I press "Add Product"
    And wait for AJAX to finish
    And I fill in "Name" with "Wireless Keyboard"
    And I fill in "Price" with "49.99"
    And I fill in "SKU" with "KB-100"
    And I select "Accessories" from "Category"
    And I press "Save"
```

```
    And wait for AJAX to finish
    Then I should see "Product created"

    When I go to "/admin/products"
    And wait for AJAX to finish
    Then I should see "Wireless Keyboard"
    And I should see "$49.99" in the "Wireless Keyboard" row
    And I should see "KB-100" in the "Wireless Keyboard" row
    And I should see "Accessories" in the "Wireless Keyboard" row

  Scenario: Edit a product from the table
    When I click "Edit" in the "Wireless Keyboard" row
    And wait for AJAX to finish
    And I fill in "Price" with "39.99"
    And I press "Save"
    And wait for AJAX to finish
    Then I should see "Product updated"

    When I go to "/admin/products"
    And wait for AJAX to finish
    Then I should see "$39.99" in the "Wireless Keyboard" row

  Scenario: Delete a product from the table
    When I click "Delete" in the "Wireless Keyboard" row
    And wait for the modal to appear
    And I click "Confirm" in the modal
    And wait for AJAX to finish
    Then I should see "Product deleted"
    And I should not see "Wireless Keyboard"
```

How It Works:

Notice the structure of the create scenario: save the form, check the success message, then navigate back to the list with `I go to "/admin/products"` and verify the row there. The re-navigation is deliberate. Many apps show an optimistic success message before the database write commits, or render the list from a cache. By forcing a fresh page load, the test reads what the backend actually persisted, not what the previous page claimed. The `in the "Wireless Keyboard" row` assertions then verify all four fields landed in one row, which catches partial saves (name persisted, category dropped) that a single `should see` misses.

One caution: these three scenarios depend on each other, edit assumes create ran, delete assumes both did. Cucumber runs scenarios in file order by default, so this works, but it breaks the moment you run scenarios in parallel or filter by tag. For a suite you will maintain long-term, prefer one longer scenario covering the full lifecycle, or give each scenario its own setup.

Pro Tip: End the lifecycle with delete even when delete is not what you are testing. A CRUD scenario that cleans up after itself can run twice in a row without "Wireless Keyboard already exists" failures, which is exactly what happens when a CI retry re-runs the feature.

Recipe 92 Testing Sortable Tables

Problem: A customer reports that sorting the product list by price puts a $9.99 cable between two $900 laptops, the column sorts alphabetically because the backend treats price as a string. Nobody caught it because nobody tests sorting; it is the kind of feature that "obviously works" until it does not. You want a scenario that clicks each sortable header and proves the order actually changed, without turning into a brittle position-by-position comparison.

Solution:

```
Feature: Sortable Table
  As a user viewing a list of products
  I want to sort by different columns
  So that I can find what I need quickly

  Background:
    Given I am a logged in user with the "admin" user
    And I go to "/admin/products"
    And wait for AJAX to finish

  Scenario: Sort by name ascending
    When I click "Name"
    And wait for AJAX to finish
    Then I should see "Apple Watch"
    # First alphabetical item should be visible at the top

  Scenario: Sort by name descending
    When I click "Name"
    And wait for AJAX to finish
    # Click again to reverse sort
    When I click "Name"
    And wait for AJAX to finish
    Then I should see "Wireless Mouse"
    # Last alphabetical item should now be at the top

  Scenario: Sort by price
    When I click "Price"
    And wait for AJAX to finish
    Then I should see "$9.99"
    # Cheapest item should be visible

  Scenario: Sort by date
    When I click "Created"
    And wait for AJAX to finish
    Then I should see "2024-01-01"
    # Oldest item should be visible
```

How It Works:

Sortable tables typically use `<th>` elements as click targets. `When I click "Name"` finds the `<a>` or clickable element with text “Name” in the table header and clicks it. The server (or JavaScript) re-orders the rows, and the `wait for AJAX` ensures the new order has loaded. If the sort is server-side, that step watches the fetch/XHR counter drop back to zero; if it is client-side JavaScript re-ordering existing rows, the auto-settle behavior after each step still waits for the DOM mutations to go quiet before the next assertion runs, so the same scenario works against both implementations without you knowing which one the developers chose.

The assertions lean on a practical trick: with a paginated table, the first page after sorting ascending contains the alphabetically-first item, and after the second click (descending) it contains the last. If “Wireless Mouse” is visible on page one after a descending sort, the sort direction flipped, and the string-vs-number price bug from the Problem above shows up immediately, because `$9.99` will not be on the first page of a correctly ascending numeric sort.

Sort Interaction Flow:

```
<thead>
   <tr>
      <th><a>Name      </a></th>        ◀ Click here
      <th><a>Price</a></th>
      <th><a>Created</a></th>
   </tr>
</thead>
<tbody>
   <tr><td>Apple Watch</td>...</tr>                ◀ First
   <tr><td>Laptop Pro</td>...</tr>
   <tr><td>Wireless Mouse</td>...</tr>
</tbody>

After clicking "Name" again (descending):
<tbody>
   <tr><td>Wireless Mouse</td>...</tr>              ◀ First
   <tr><td>Laptop Pro</td>...</tr>
   <tr><td>Apple Watch</td>...</tr>
</tbody>
```

FIGURE 9.4 *Testing Sortable Tables (Recipe 92)*

Note: Testing exact sort order is difficult with `should see` because it only checks presence, not position. For most cases, verifying that the expected first/last item is visible after sorting is sufficient. If you need to verify exact order, use Scenario Outlines with expected positions or custom step definitions.

At scale: Sort tests only stay portable if your seed data is designed for them. Across multiple projects, seed every list fixture with three sentinel records: one that sorts first alphabetically, one that sorts last, and one with an extreme numeric value. Then the same four sort scenarios work on every project's tables, and a sort regression on any of them fails loudly instead of depending on whatever data a developer happened to have locally.

Recipe 93 Testing Filterable Tables

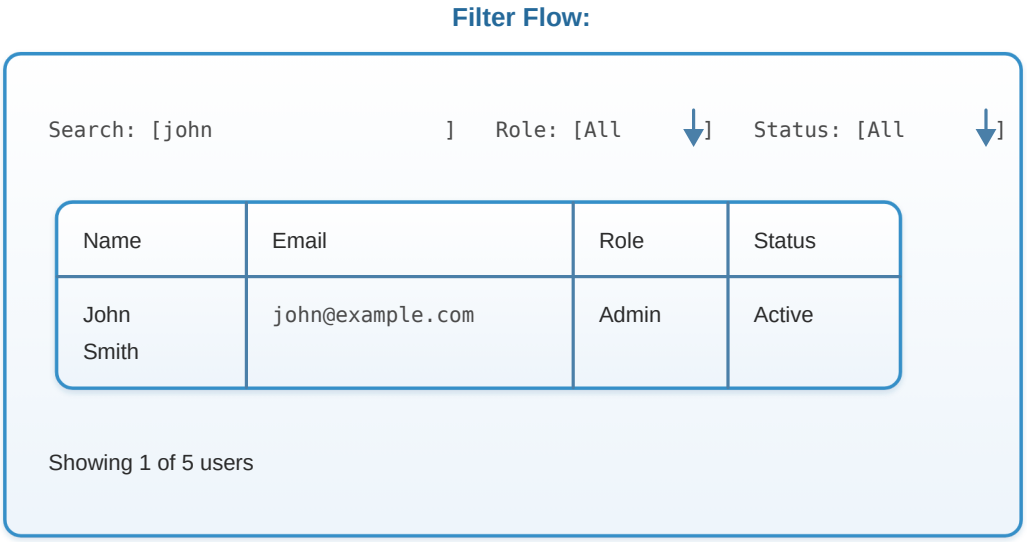
Problem: Support escalates a ticket: an admin filtered the user list by “Blocked” status, exported the result, and emailed warnings to forty accounts, half of them active, because the filter silently matched nothing and showed everyone. A filter that fails open is worse than no filter, and it is invisible to a test that only checks the rows you expect to see. The day this bites you, you learn that filter tests must assert what disappeared, not just what stayed.

Solution:

```
Feature: Filterable Table
  As an admin viewing users
  I want to filter the table
  So that I can find specific users quickly

  Background:
    Given I am a logged in user with the "admin" user
    And I go to "/admin/users"
    And wait for AJAX to finish

  Scenario: Filter by search text
    When I fill in "Search" with "john"
    And wait for AJAX to finish
    Then I should see "John Smith"
    And I should see "john@example.com"
    And I should not see "Jane Doe"
    And I should not see "Bob Wilson"

  Scenario: Filter by role dropdown
    When I select "Admin" from "Role"
    And wait for AJAX to finish
    Then I should see "John Smith"
    And I should not see "Jane Doe"

  Scenario: Filter by status
    When I select "Active" from "Status"
    And wait for AJAX to finish
```

```
    Then I should see "John Smith"
    Then I should see "Jane Doe"
    And I should not see "Blocked User"

  Scenario: Combine multiple filters
    When I fill in "Search" with "john"
    And I select "Admin" from "Role"
    And wait for AJAX to finish
    Then I should see "John Smith"
    And I should not see "Jane Doe"

  Scenario: Clear filters shows all data
    When I fill in "Search" with "john"
    And wait for AJAX to finish
    Then I should not see "Jane Doe"

    When I fill in "Search" with ""
    And wait for AJAX to finish
    Then I should see "John Smith"
    And I should see "Jane Doe"

  Scenario: No results found
    When I fill in "Search" with "zzzznonexistent"
    And wait for AJAX to finish
    Then I should see "No results found"
    And I should not see "John Smith"
```

How It Works:



The filter step triggers either:
1. Client-side filtering (JavaScript hides non-matching rows)
2. Server-side filtering (AJAX request fetches filtered data)

Either way, "wait for AJAX to finish" handles the timing.

FIGURE 9.5 *Testing Filterable Tables (Recipe 93)*

`I fill in "Search" with "john"` resolves the field label-first, it looks for a `<label>` reading "Search", then falls back to id, name, and placeholder attributes. That matters for filters because filter inputs are often placeholder-only ("Search users…"); the step still finds them without a CSS selector. After typing, the smart-wait machinery takes over: whether the app debounces and fires an AJAX request or filters rows with pure JavaScript, the settle check (no pending fetch/XHR, no pending timers, DOM quiet for 250ms) holds the next step until the table has finished re-rendering.

Tip: When testing filters, always verify both what SHOULD appear and what SHOULD NOT appear. Just checking "I should see John" does not prove the filter worked, John might have been visible anyway. The `should not see` assertion for excluded items is what proves the filter is

actually filtering. The “fails open” bug in the Problem above passes every positive assertion and fails only the negative one.

At scale: The combine-multiple-filters scenario is the one teams skip and the one that breaks most often, because each filter is usually developed and reviewed in isolation. When you maintain table tests across many projects, make “one combined-filter scenario per filterable table” a checklist item in your test plan template. It is one scenario per table, and it is the only automated evidence that filters compose correctly, the kind of line item a QA lead can point to when a release manager asks what “tested” means.

Recipe 94 Testing Table Pagination

Problem: Your order list paginates at ten rows, and for months that was enough, staging never had more than eight orders. Then production crosses the threshold and the “Next” button 404s, or page two shows the same ten orders as page one because the offset parameter was never wired up. Pagination bugs hide in exactly the data volumes your test environment does not have, which is why they ship. The fix is a seeded dataset big enough to paginate, plus scenarios that walk the boundaries.

Solution:

```
Feature: Table Pagination
  As an admin viewing a long list
  I want to navigate between pages
  So that I can see all records

  Background:
    Given I am a logged in user with the "admin" user
    And I go to "/admin/orders"
    And wait for AJAX to finish

  Scenario: First page shows first batch of data
    Then I should see "ORD-001"
    And I should see "ORD-010"
    And I should not see "ORD-011"
```

```
  And I should see "Showing 1-10 of 50"

Scenario: Navigate to next page
  When I click "Next"
  And wait for AJAX to finish
  Then I should see "ORD-011"
  And I should see "ORD-020"
  And I should not see "ORD-001"
  And I should see "Showing 11-20 of 50"

Scenario: Navigate to specific page
  When I click "3"
  And wait for AJAX to finish
  Then I should see "ORD-021"
  And I should see "Showing 21-30 of 50"

Scenario: Navigate to previous page
  When I click "2"
  And wait for AJAX to finish
  Then I should see "ORD-011"
  When I click "Previous"
  And wait for AJAX to finish
  Then I should see "ORD-001"
  And I should see "Showing 1-10 of 50"

Scenario: Change items per page
  When I select "25" from "Items per page"
  And wait for AJAX to finish
  Then I should see "ORD-001"
  And I should see "ORD-025"
  And I should see "Showing 1-25 of 50"

Scenario: Last page navigation
  When I click "Last"
  And wait for AJAX to finish
  Then I should see "ORD-050"
  And I should not see "Next"
```

How It Works:

The scenarios use sequential reference numbers (ORD-001 through ORD-050) as both row content and boundary markers. That gives every page a predictable first and last record, so each scenario asserts three things: the new page's first item is present, the old page's items are gone (`should not see "ORD-001"` after clicking Next), and the "Showing X-Y of Z" counter agrees. The counter assertion is the cheapest of the three and catches a surprising number of off-by-one bugs, a page that shows eleven rows, or a total count that ignores filters.

The duplicate-page bug from the Problem statement is caught by the negative assertion. If page two re-renders page one's data, `I should see "ORD-011"` fails immediately. Without the negative check, a broken offset that appends rows instead of replacing them would pass.

At scale: Pagination tests demand seeded volume, at least two and a half pages of records, created by your seed script with predictable identifiers. Across many projects, put the record count and the per-page size into the seed configuration rather than hardcoding them into scenarios, and standardize one page size across your portfolio's admin UIs if you can. The same six pagination scenarios then run everywhere, and the only thing each project supplies is its own fixture file.

Recipe 95 Testing Table Search

Problem: The product search box works perfectly in every demo, and then a flaky test report lands on your desk: search scenarios pass locally, fail one run in five on CI. The culprit is the debounce. The app waits 300ms after the last keystroke before firing the request, and on a slow CI runner your assertion sometimes runs in that gap, against the unfiltered table. You need scenarios that test the search behavior itself, matching, case-insensitivity, empty results, clearing, without racing the debounce timer.

Solution:

```
Feature: Table Search
  Background:
    Given I am a logged in user with the "admin" user
    And I go to "/admin/products"
```

```
  And wait for AJAX to finish

Scenario: Search finds matching products
  When I fill in "Search products" with "laptop"
  And wait for AJAX to finish
  Then I should see "Laptop Pro"
  And I should see "Laptop Air"
  And I should not see "Wireless Mouse"

Scenario: Search is case insensitive
  When I fill in "Search products" with "LAPTOP"
  And wait for AJAX to finish
  Then I should see "Laptop Pro"

Scenario: Search with no results
  When I fill in "Search products" with "xyznonexistent"
  And wait for AJAX to finish
  Then I should see "No products found"

Scenario: Clear search restores all products
  When I fill in "Search products" with "laptop"
  And wait for AJAX to finish
  Then I should not see "Wireless Mouse"

  When I fill in "Search products" with ""
  And wait for AJAX to finish
  Then I should see "Wireless Mouse"
  And I should see "Laptop Pro"

Scenario: Search with debounce
  When I fill in "Search products" with "wire"
  # Wait for the debounce delay (typically 300ms)
  And wait 1s
  And wait for AJAX to finish
  Then I should see "Wireless Mouse"
  And I should see "Wireless Keyboard"
```

How It Works:

Here is the part most people miss: webship-js already knows about the debounce. Through `addInitScript`, it wraps `setTimeout` on every page and counts pending timers in `window.__webshipPendingTimers`, alongside the fetch/XHR counter in `window.__webshipAjaxCount`. When the app schedules a 300ms debounce after your keystroke, that timer counts as "pending", and the settle check that runs after each step (ajax count zero AND pending timers zero AND DOM quiet for at least 250ms) will not release the next step until the debounce fires, the request completes, and the table re-renders. The race condition from the Problem statement is exactly what this machinery exists to kill.

So in most suites the explicit `wait 1s` in the last scenario is redundant, the auto-settle covers it. It earns its keep only for apps that debounce with something the timer wrapper cannot see (a requestAnimationFrame loop, a third-party widget with its own scheduler). Try removing it; if the scenario stays green for a week of CI runs, leave it out.

The case-insensitivity scenario is worth keeping even though it looks trivial. Search case-sensitivity depends on the database collation, not the application code, so it is one of the few behaviors that genuinely differs between environments, a SQLite-backed dev setup and a PostgreSQL production can disagree.

Recipe 96 Testing Bulk Actions

Problem: Bulk actions are where a small bug becomes a big incident. An admin selects three products to archive, the checkbox state desyncs after a filter change, and "Apply" archives the whole catalog. These are the most destructive buttons in your application, usually behind the least-tested code path, and the confirmation modal that is supposed to be the safety net often displays the wrong count. If you automate only one thing on an admin table, automate this.

Solution:

```
Feature: Bulk Table Actions
  As an admin
  I want to perform actions on multiple rows at once
```

```
So that I can manage data efficiently

Background:
  Given I am a logged in user with the "admin" user
  And I go to "/admin/products"
  And wait for AJAX to finish

Scenario: Select all rows and delete
  When I check "Select All"
  And I press "Delete Selected"
  And wait for the modal to appear
  Then I should see "Delete 10 products?" in the modal
  When I click "Confirm" in the modal
  And wait for AJAX to finish
  Then I should see "10 products deleted"

Scenario: Select individual rows
  When I check "Laptop Pro"
  And I check "Wireless Mouse"
  And I press "Delete Selected"
  And wait for the modal to appear
  Then I should see "Delete 2 products?" in the modal
  When I click "Cancel" in the modal
  And wait for the modal to disappear
  # Products should still be there
  Then I should see "Laptop Pro"
  And I should see "Wireless Mouse"

Scenario: Bulk status change
  When I check "Select All"
  And I select "Archive" from "Bulk Actions"
  And I press "Apply"
  And wait for AJAX to finish
  Then I should see "10 products archived"

Scenario: Export selected rows
  When I check "Laptop Pro"
  And I check "Wireless Mouse"
  And I check "USB Cable"
```

```
And I press "Export Selected"
And wait for AJAX to finish
Then I should see "3 products exported"
```

How It Works:

`When I check "Laptop Pro"` uses the same label-first resolution as form fields: it looks for a checkbox whose associated label (or nearby text) matches “Laptop Pro”. Most row-selection checkboxes are wired to the row’s title through a label or aria-label, so this works without selectors. If your table’s checkboxes are anonymous (`<input type="checkbox">` with no label), ask the developers to add aria-labels, it fixes your tests and their accessibility audit in one change.

The real assertions in these scenarios are the counts. `"Delete 10 products?"` in the modal and `"10 products deleted"` afterward verify two different things: the first proves the selection state was correct *before* the destructive action, the second proves the backend acted on exactly that selection. The checkbox-desync bug from the Problem statement fails the first assertion, which is the one that fires before any data is harmed. And note the Cancel scenario: proving that Cancel does nothing is as important as proving Confirm does something.

At scale: Tag these scenarios (something like `@destructive`) and make your CI seed a fresh dataset before they run, a bulk-delete scenario that runs against shared test data will sabotage every scenario after it. On a multi-project CI farm this is a pipeline-ordering question for your DevOps team: destructive suites run last, or against their own database snapshot, and never against an environment another team is mid-run on.

Recipe 97 Testing Expandable/Collapsible Rows

Problem: The orders table got a nice UX upgrade: click a row and it expands to show the line items inline, no page navigation. Then a JavaScript refactor breaks the toggle in one direction, rows expand but never collapse, or the second click navigates to a detail page instead. Nobody notices in review because everyone tests the expand, never the collapse. Toggle interactions need both directions tested, every time.

Solution:

```
Feature: Expandable Table Rows
  As a user viewing orders
  I want to expand rows for details
  So that I can see order items without leaving the list

  Background:
    Given I am a logged in user with the "admin" user
    And I go to "/admin/orders"
    And wait for AJAX to finish

  Scenario: Expand a row to see details
    Then I should see "ORD-001"
    And I should not see "Order Items"

    When I click "ORD-001"
    And wait 1s
    Then I should see "Order Items"
    And I should see "Laptop Pro x 1"
    And I should see "USB Cable x 2"

  Scenario: Collapse an expanded row
    When I click "ORD-001"
    And wait 1s
    Then I should see "Order Items"

    When I click "ORD-001"
    And wait 1s
    Then I should not see "Order Items"

  Scenario: Multiple rows can be expanded
    When I click "ORD-001"
    And wait 1s
    When I click "ORD-002"
    And wait 1s
    Then I should see "Laptop Pro x 1"
    And I should see "Wireless Mouse x 3"
```

How It Works:

The first scenario sets up the assertion before the action: `I should not see "Order Items"` confirms the detail panel is genuinely hidden before the click, so the post-click `I should see` proves the click caused the change. Without that baseline, an expand panel that renders open by default would pass the test while the toggle is broken.

The `wait 1s` after each click is there for the expand animation. If the row content is inserted into the DOM by JavaScript, webship-js's MutationObserver (tracked through `window.__webshipLastMutation`) already holds the next step until the DOM has been quiet for 250ms, so for most implementations the auto-settle covers it and the explicit wait can go. Where it still helps is CSS-only expansion: content that is already in the DOM but revealed by a height transition produces no mutations and no AJAX, so nothing tells webship-js to wait. Know which kind your app uses before deleting the wait.

The multi-expand scenario answers a product question, not just a technical one: is this an accordion (opening one row closes the other) or independent panels? Either behavior can be correct, but the test pins down which one was intended, so a refactor that silently changes it gets caught.

Recipe 98 Testing Empty State Tables

Problem: Your demo environment always has data, so nobody on the team has seen the orders page the way a brand-new customer sees it: empty. Then the first real signup lands on a bare table header with zero rows, or worse, a JavaScript error because the rendering code assumed at least one record. Empty states are the first screen every new user meets, and the last screen anyone on the team looks at. That mismatch is exactly why they break.

Solution:

```
Feature: Empty Table State
  Scenario: New account has no orders
    Given I am a logged in user with the "newuser" user
    When I go to "/orders"
    And wait for AJAX to finish
    Then I should see "No orders yet"
    And I should see "Start shopping to see your orders here"
    And I should see "Browse Products"

  Scenario: Filter produces no results
    Given I am a logged in user with the "admin" user
    When I go to "/admin/users"
    And wait for AJAX to finish
    And I fill in "Search" with "nonexistentuser12345"
    And wait for AJAX to finish
    Then I should see "No users found"
    And I should see "Try different search terms"

  Scenario: Empty state has call to action
    Given I am a logged in user with the "admin" user
    When I go to "/admin/categories"
    And wait for AJAX to finish
    Then I should see "No categories yet"
    And I should see "Create your first category"
    When I press "Create Category"
    And wait for AJAX to finish
    Then I should see "New Category"
```

How It Works:

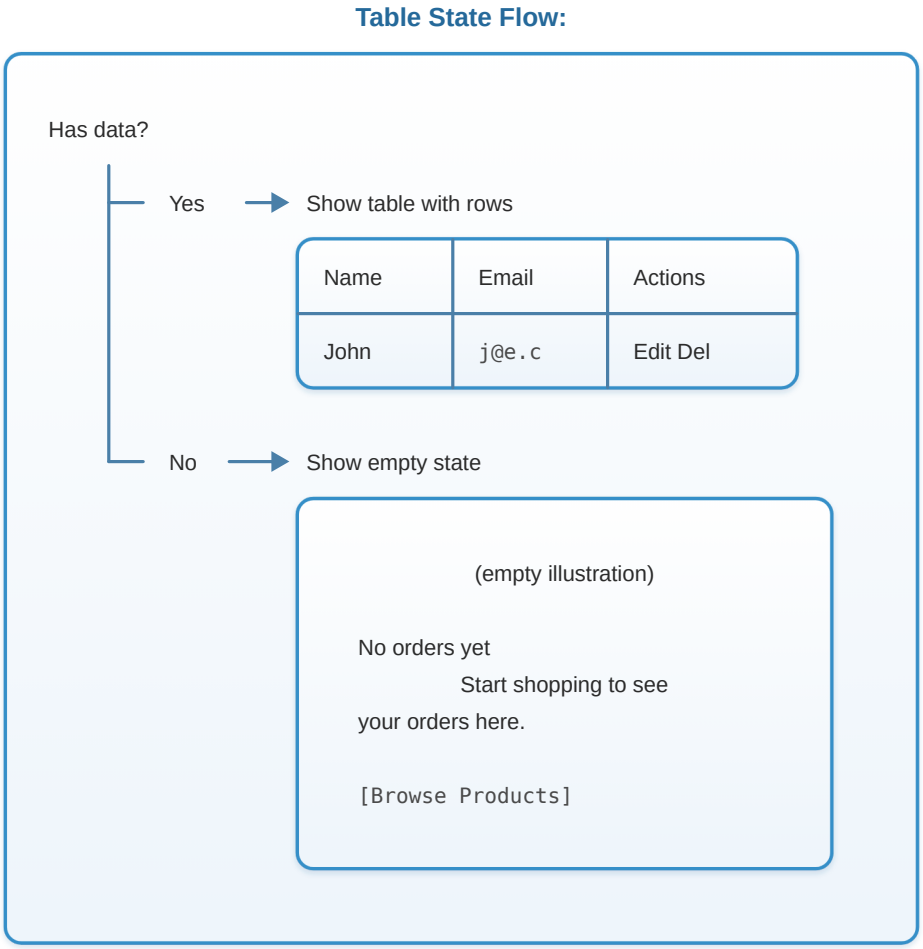


FIGURE 9.6 *Testing Empty State Tables (Recipe 98)*

Empty states are important to test because:

- They are the first thing new users see
- A broken empty state creates confusion ("Is the page loading? Is it broken?")
- The call-to-action button guides users to the next step
- They should never show a broken table with zero rows and no explanation

The three scenarios cover the three distinct ways a table ends up empty: a genuinely new account (`newuser`, which means your seed data must include a user with no records, a fixture most teams forget to create), a filter that matches nothing, and an empty collection with a creation flow. They are different code paths in most apps, the “no data at all” branch and the “no data matching your filter” branch are usually separate templates, so one passing scenario does not vouch for the others.

At scale: Empty-state rendering is where unhandled JavaScript errors love to hide (`cannot read property 'length' of undefined`). Run these scenarios with JS error capture on, `WEBSHIP_JS_ERROR_MODE=fail`, or tag the scenarios `@js-fail`, so a console error on the empty page fails the test even when the visible message happens to render. That single setting turns three modest scenarios into a crash detector for every new account’s first session.

Recipe 99 Testing Inline Editing in Tables

Problem: Inline editing is a gift to admins and a trap for testers. The price cell turns into an input, the admin types a new value, hits Save, and sometimes the cell shows the new price while the database still holds the old one, because the optimistic UI updated before the failed request came back. Your test clicks Save, sees “$1,099.99”, and goes green on a value that never persisted. Inline edit tests have to distrust what the cell shows immediately after saving.

Solution:

```
Feature: Inline Table Editing
  As an admin
  I want to edit values directly in the table
  So that I can make quick changes without opening a form

  Background:
    Given I am a logged in user with the "admin" user
    And I go to "/admin/products"
    And wait for AJAX to finish

  Scenario: Edit price inline
    When I click "Edit" in the "Laptop Pro" row
    And wait 1s
    And I fill in "Price" with "1099.99"
    And I press "Save"
    And wait for AJAX to finish
    Then I should see "$1,099.99" in the "Laptop Pro" row

  Scenario: Edit status inline
    When I click "Edit" in the "Laptop Pro" row
    And wait 1s
    And I select "Out of Stock" from "Status"
    And I press "Save"
    And wait for AJAX to finish
    Then I should see "Out of Stock" in the "Laptop Pro" row

  Scenario: Cancel inline edit
    When I click "Edit" in the "Laptop Pro" row
    And wait 1s
    And I fill in "Price" with "0.01"
    And I press "Cancel"
    And wait 1s
    Then I should see "$999.99" in the "Laptop Pro" row
    And I should not see "$0.01"
```

How It Works:

When the Edit click swaps the cell into edit mode, the new `<input>` is inserted by JavaScript, and `I fill in "Price"` finds it by label-first resolution the same as any form field, falling back to id, name, and placeholder. Inline editors frequently skip visible labels, so check what the input actually carries; a `placeholder="Price"` is enough for the step to resolve it. After Save, the AfterStep settle waits for the PATCH/POST to complete before the row assertion runs.

To close the optimistic-UI gap from the Problem statement, do not stop at asserting the cell text after Save. Add a `When I reload the page` followed by the same row assertion: the reload forces a fresh read from the backend, so the test verifies what persisted rather than what the JavaScript painted. It costs one extra page load and converts a cosmetic check into a data-integrity check.

The Cancel scenario deserves its place: it asserts both that the old value survived ($999.99) and that the typed-but-discarded value left no trace ($0.01). An inline editor that "cancels" the UI but still fires the save request is a real bug class, and only the follow-up assertions catch it.

Recipe 100 Testing Drag-and-Drop Reordering

Problem: The menu builder lets editors drag rows to reorder navigation, and the ticket says "automate it." Two weeks later the drag test passes on Chromium, fails on WebKit, and passes-or-fails on Firefox depending on the runner's mood, because HTML5 drag events, mouse event timing, and the drag library's own thresholds all interact differently per browser. Meanwhile the actual business requirement ("editors can change the menu order and it persists") could have been green on day one through the Move Up/Move Down buttons.

Solution:

Drag-and-drop requires a custom step definition since webship-js does not include a built-in drag step:

```
Feature: Table Row Reordering
  Scenario: Reorder items by drag and drop
    Given I am a logged in user with the "admin" user
    When I go to "/admin/menu-items"
    And wait for AJAX to finish
    Then I should see "Home"
    And I should see "About"
    And I should see "Contact"

    # Use move up/down buttons if available (easier to test)
    When I click "Move Up" in the "Contact" row
    And wait for AJAX to finish
    Then I should see "Order saved"

  Scenario: Reorder with up/down buttons
    Given I am a logged in user with the "admin" user
    When I go to "/admin/menu-items"
    And wait for AJAX to finish
    When I click "Move Down" in the "Home" row
    And wait for AJAX to finish
    Then I should see "Order updated"
```

Custom Step Definition for Drag-and-Drop:

```
// tests/step-definitions/custom-drag.js
const { When } = require('@cucumber/cucumber');

When('I drag {string} to {string}', async function (source, target) {
  const sourceEl = this.page.getByText(source).first();
  const targetEl = this.page.getByText(target).first();

  const sourceBox = await sourceEl.boundingBox();
  const targetBox = await targetEl.boundingBox();

  await this.page.mouse.move(
    sourceBox.x + sourceBox.width / 2,
    sourceBox.y + sourceBox.height / 2
  );
  await this.page.mouse.down();
  await this.page.mouse.move(
    targetBox.x + targetBox.width / 2,
    targetBox.y + targetBox.height / 2,
    { steps: 10 }
  );
  await this.page.mouse.up();
});
```

How It Works:

The custom step drives raw mouse events through Playwright's mouse API: move to the source row's center, press down, glide to the target in ten intermediate steps, release. The `steps: 10` option is not decoration, drag libraries typically ignore a teleporting cursor and only engage after seeing continuous movement past a threshold, so the intermediate move events are what make the library believe a human is dragging. Even then, libraries built on HTML5 drag events (`dragstart` / `drop`) rather than mouse events will not respond to this at all; those need `dispatchEvent`-based workarounds that are even more fragile.

This is why the Gherkin scenarios lean on Move Up/Move Down buttons instead. The buttons hit the same backend reorder endpoint and the same persistence logic, the part that actually matters to the business, through a click step that is reliable on every browser. The drag gesture itself becomes a thin UI layer you verify manually or in a single quarantined scenario.

Pro Tip: Drag-and-drop is notoriously difficult to test reliably across browsers and frameworks. If your application supports both drag-and-drop and button-based reordering (move up/move down), test the button-based approach in your automated tests. It is more reliable and tests the same business logic. Reserve drag-and-drop testing for manual testing or a dedicated drag-drop test with the custom step definition above.

At scale: If several of your projects need drag testing, do not let each suite grow its own copy of this custom step. Put it in a shared step-definition package your projects install, and tag every scenario that uses it (say `@drag`) so CI can run those scenarios on Chromium only and skip them on the browsers where they flake. One shared implementation plus one tag convention beats five slightly-different copies producing five different flake patterns in your nightly reports.

Testing Lists (Non-Table Patterns)

Not all data lists use `<table>` elements. Many modern applications use card layouts, list groups, or grid layouts. The same principles apply:

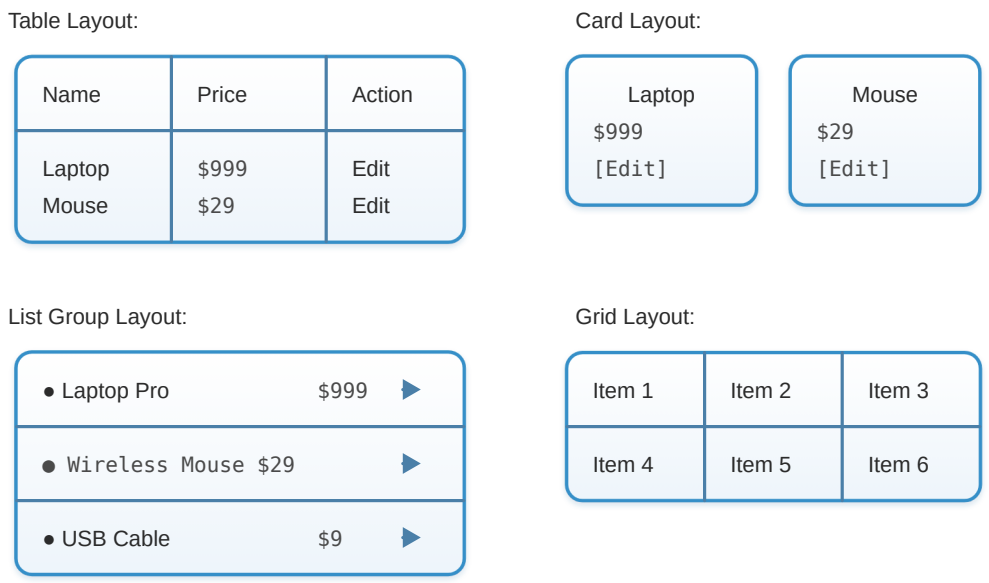


FIGURE 9.7 *Testing Lists (Non-Table Patterns)*

Recipe 101 Testing Card-Based Lists

Problem: The design team replaces the products table with a card grid, and overnight half your table tests stop making sense, there are no `<tr>` elements left, so every `in the "X" row` step fails with “row not found”. The data and the behavior have not changed; only the markup has. This is the third project where it happens, and each time someone rewrites the tests from scratch. There is a cheaper way through.

Solution:

```
Feature: Card List
  Scenario: Product cards display correctly
    Given I am on homepage
    When I go to "/products"
    And wait for AJAX to finish
    Then I should see "Laptop Pro"
    And I should see "$999"
    And I should see "Wireless Mouse"
    And I should see "$29"

  Scenario: Click a card to view details
    Given I am on homepage
    When I go to "/products"
    And wait for AJAX to finish
    And I click "Laptop Pro"
    And wait for AJAX to finish
    Then I should see "Laptop Pro"
    And I should see "Detailed specifications"
    And I should see "Add to Cart"

  Scenario: Card actions
    Given I am a logged in user with the "admin" user
    When I go to "/admin/products"
    And wait for AJAX to finish
    And I press "Edit"
    And wait for AJAX to finish
    Then I should see "Edit Product"
```

How It Works:

For card-based layouts, the `in the "X" row` steps will not work because there are no `<tr>` elements, the row steps query `tr` specifically and filter by text, so a `<div class="card">` is invisible to them. Instead, use the general `should see`, `click`, and `press` steps. If you need to interact with a specific card's action button, two approaches:

1. **Click the card text first** to navigate to its detail page, then interact there
2. **Use custom step definitions** that scope actions to a card container:

```
// tests/step-definitions/custom-card.js
const { When, Then } = require('@cucumber/cucumber');

When('I click {string} on the {string} card', async function (action,
cardName) {
  const card = this.page.locator('.card, .product-card, [class*="card"]')
    .filter({ hasText: cardName })
    .first();
  await card.locator('a, button, [role="button"]')
    .filter({ hasText: action })
    .first()
    .click();
});
```

The custom step mirrors the row-step strategy exactly, find the container by its text, then click inside it, it just swaps `tr` for the card container's selector. That symmetry is the lesson: scoped interaction is a pattern, not a table feature.

At scale: The fragile part of the custom step is the container selector (`.card, .product-card, [class*="card"]`), every project's design system names it differently. Keep that selector out of the step code: register it as a named selector in a per-project JSON file under `tests/selectors/` (loaded automatically through `worldParameters.selectors.files`), and have the shared step look the name up. Then twenty projects share one step definition, and onboarding a new project means writing one JSON entry, not forking a step file.

Recipe 102 Testing Data Table with Scenario Outline

Problem: Verifying four users means eight assertions; verifying forty means eighty, and the feature file turns into copy-paste sludge nobody reviews carefully. Then a product owner asks “which users do we actually verify?” and the honest answer is buried across three hundred lines of near-identical scenarios. You want the expected data in one readable table that a non-engineer can review in a pull request, and the repetition generated for you.

Solution:

```
Feature: Data Verification
  Background:
    Given I am a logged in user with the "admin" user
    And I go to "/admin/users"
    And wait for AJAX to finish

  Scenario Outline: User data is correct in table
    Then I should see "<role>" in the "<email>" row
    And I should see "<status>" in the "<email>" row

    Examples:
      | email              | role   | status  |
      | john@example.com   | Admin  | Active  |
      | jane@example.com   | Editor | Active  |
      | bob@example.com    | Viewer | Blocked |
      | alice@example.com  | Admin  | Active  |

  Scenario Outline: Action buttons are available for each user
    When I click "View" in the "<email>" row
    And wait for AJAX to finish
    Then I should see "<name>"
    When I move backward one page

    Examples:
      | email              | name        |
      | john@example.com   | John Smith  |
      | jane@example.com   | Jane Doe    |
      | bob@example.com    | Bob Wilson  |
```

How It Works:

Scenario Outline generates one test scenario per row in the Examples table. This is powerful for table verification because you can test every row without writing repetitive scenarios.

```
Scenario Outline generates:

Row 1: Then I should see "Admin" in the "john@example.com" row
        And I should see "Active" in the "john@example.com" row

Row 2: Then I should see "Editor" in the "jane@example.com" row
        And I should see "Active" in the "jane@example.com" row

Row 3: Then I should see "Viewer" in the "bob@example.com" row
        And I should see "Blocked" in the "bob@example.com" row

Row 4: Then I should see "Admin" in the "alice@example.com" row
        And I should see "Active" in the "alice@example.com" row
```

Each generated scenario is independent: it gets its own Background run (login, navigate, settle) and reports its own pass/fail line. That granularity is the win over one giant scenario with eight assertions, when bob@example.com's role regresses, the report says exactly which example failed, instead of failing one mega-scenario and hiding the other seven results behind it. The price is runtime: four examples means four logins and four page loads. Keep outlines for data variation; do not use them to re-run identical behavior.

There is a quieter benefit for the people who do not write tests. The Examples table *is* the acceptance criteria, a product owner can read it, spot that alice should be an Editor not an Admin, and comment on the pull request without parsing a line of automation code. When your BA writes the Examples table and QA writes the steps above it, sign-off stops being an act of trust.

At scale: The Examples table is data, and data wants to live with the seed fixtures. If the same user matrix is seeded in every project, generate or copy the Examples block from the same source the seed script uses, one YAML or JSON file feeding both. The moment seed data and Examples tables are maintained by hand in two places, they will drift, and you will spend an afternoon debugging a "failure" that is just a stale expectation.

Table Testing Quick Reference

WHAT YOU WANT	STEP TO USE
Verify text exists on page	Then I should see "text"
Verify text in specific row	Then I should see "text" in
	the "identifier" row
Verify text NOT in a row	Then I should not see "text"
	in the "identifier" row
Click action in a row	When I click "Edit" in the
	"identifier" row
Sort by column	When I click "Column Name"
Filter by text	When I fill in "Search"
	with "query"
Filter by dropdown	When I select "value" from
	"filter"
Navigate to next page	When I click "Next"
Select a row (checkbox)	When I check "identifier"
Select all rows	When I check "Select All"
Empty state	Then I should see "No data"
Wait after table action	And wait for AJAX to finish

Chapter Summary

- Tables are the core of data-driven applications
- Use `in the "X" row` steps to scope assertions and actions to specific rows
- Choose unique row identifiers: emails, IDs, reference numbers
- Always verify both presence AND absence when testing filters
- Test the complete CRUD lifecycle through the table
- Sort testing verifies the first/last visible item after sorting
- Pagination testing navigates forward, backward, and to specific pages
- Bulk actions: select rows → choose action → confirm → verify result
- Empty states should show helpful messages with calls to action
- Card layouts need different strategies than table layouts
- Use Scenario Outline + Examples for efficient multi-row verification
- Always `wait for AJAX to finish` after actions that trigger server requests

Next: Authentication and Authorization Testing recipes.

Chapter 10: Authentication and Authorization Testing

Almost every web application has a login page. And behind that login page sits an entire authorization system that determines what each user can see, click, and modify. A user with the “admin” role sees a different dashboard than a user with the “viewer” role. An unauthenticated visitor gets redirected to the login page. A session expires after 30 minutes of inactivity.

These are not edge cases. They are the core of your application’s security model. A missing permission check can expose sensitive data. A broken session handler can lock users out. A redirect loop after login can make your application unusable.

This chapter teaches you how to test every authentication and authorization pattern you will encounter.

Understanding Auth in Web Applications

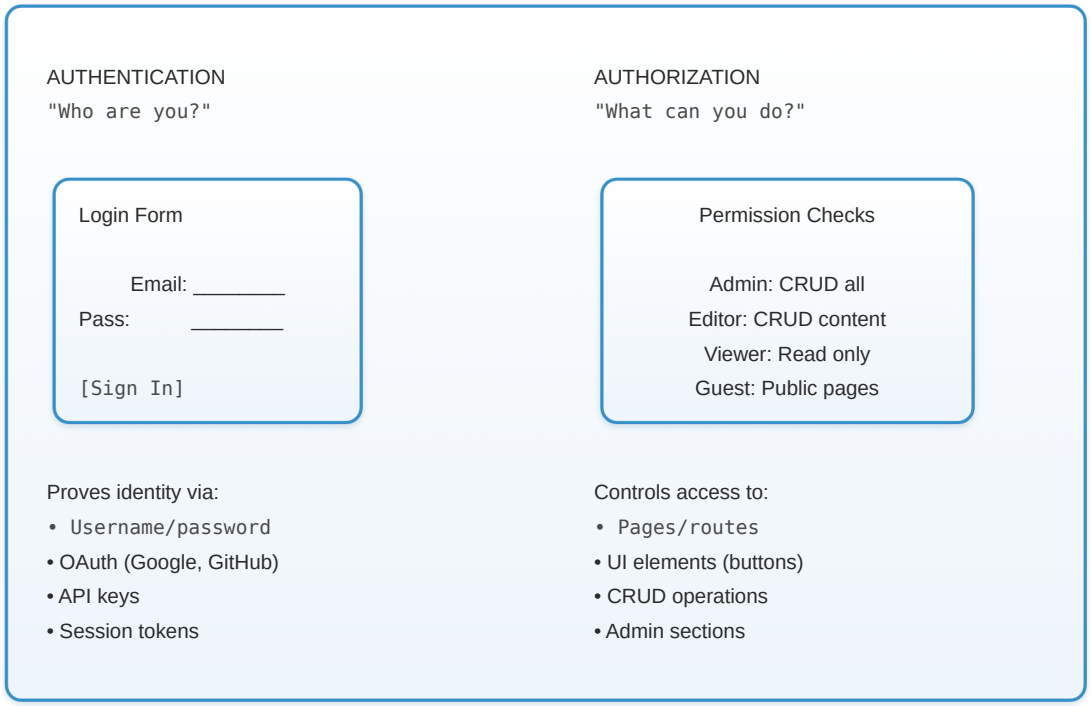


FIGURE 10.1 *Understanding Auth in Web Applications*

The testing approach differs for browser-based UI tests and API tests:

UI TESTING (PLAYWRIGHT)	API TESTING (AXIOS)
Fill login form + press submit	Basic Auth header
Browser stores session cookie	Bearer token header
Subsequent pages are authenticated	Each request needs auth
Given I am a logged in user with	Given I am authenticating
the "admin" user	as "admin" with
	"password" password
Given I am on homepage	Given I set header
When I go to "/login"	"Authorization"
And I fill in "Email" with "..."	with value "Bearer ..."
And I fill in "Password" with ".."	
And I press "Sign In"	When I send a GET request
And wait for AJAX to finish	to "/api/users"

Recipe 103 Testing Successful Login

Problem: Login is the one page every user crosses and the one your team stopped looking at months ago. Then a frontend dependency upgrade renames a form field, or an SSO change breaks the redirect, and on Monday morning nobody can get in, the whole application is down as far as your users can tell, while every other test in your suite still passes because they bypass the form. The login form needs its own test, run on every build, before anything else.

Solution:

```
Feature: User Login
  As a registered user
  I want to log in to my account
  So that I can access my personal dashboard

  Scenario: Login with valid credentials
    Given I am on homepage
    When I go to "/login"
    Then I should see "Sign In"
    When I fill in "Email" with "admin@example.com"
    And I fill in "Password" with "AdminPass123"
    And I press "Sign In"
    And wait for AJAX to finish
    Then I should see "Dashboard"
    And I should see "Welcome, Admin"
    And I should not see "Sign In"

  Scenario: Login with username instead of email
    Given I am on homepage
    When I go to "/login"
    And I fill in "Username" with "admin"
    And I fill in "Password" with "AdminPass123"
    And I press "Sign In"
    And wait for AJAX to finish
    Then I should see "Dashboard"
```

How It Works:

This is a straightforward form submission test. The flow is:

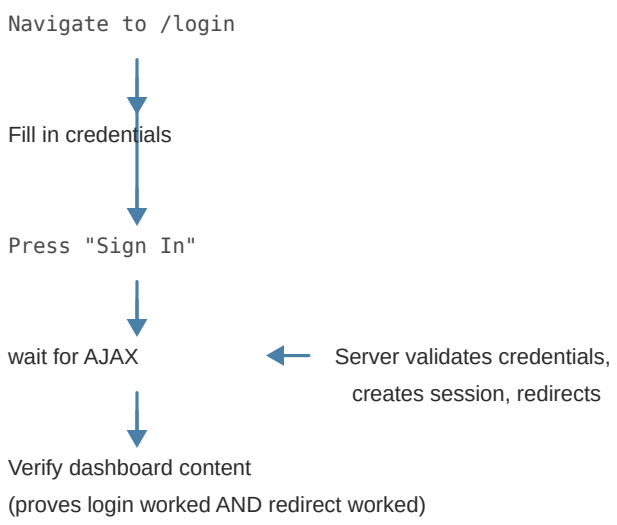


FIGURE 10.2 *Testing Successful Login (Recipe 103)*

`I fill in "Email" with "..."` resolves the field label-first, it tries the `<label>` text, then id, name, and placeholder attributes. That is why the same scenario survives a markup refactor: as long as the field is still labeled “Email” for humans, the step finds it. After pressing “Sign In”, the submit triggers a navigation or an XHR depending on how the app is built; webship-js’s settle logic covers both, waiting until in-flight requests drain and the DOM stops mutating before the dashboard assertions run. There is one deliberate redundancy in the assertions: `I should not see "Sign In"` catches the nasty failure mode where the server re-renders the login form with no error message, “Dashboard” might appear in a nav link even on the login page, but the absence check cannot be fooled that way.

Pro Tip: Do not assert the URL after login. Assert the content. If you see "Dashboard" and "Welcome, Admin," you know:

1. Login succeeded
2. Session was created
3. Redirect happened
4. Dashboard rendered with user-specific content

That is four assertions in one `should see` check. URL checking adds fragility without value.

At scale: This scenario belongs in every project's smoke tag (`@smoke` or `@critical`), and the credentials belong in environment variables, never in the feature file of a real project. When you run dozens of suites against dev, staging, and production-like environments, each environment has its own test account, wire the credentials through CI secrets and a custom login step (Recipe 105), and the same feature file runs unchanged everywhere. A login scenario that needs editing per environment is a login scenario that eventually tests the wrong environment.

Recipe 104 Testing Login Failures

Problem: A security review flags your login page: it says "User not found" for unknown emails and "Wrong password" for known ones, which means anyone can enumerate your customer list one email at a time. The developers fix it, and six months later a helpful refactor brings the friendly messages back, because nothing was guarding the behavior. Rejection paths are security requirements, and security requirements that are not pinned by a test do not stay fixed.

Solution:

```
Feature: Login Failure Handling
  As a security-conscious application
  I want to reject invalid credentials
  So that unauthorized users cannot access accounts

  Scenario: Wrong password
    Given I am on homepage
```

```
  When I go to "/login"
  And I fill in "Email" with "admin@example.com"
  And I fill in "Password" with "WrongPassword"
  And I press "Sign In"
  And wait for AJAX to finish
  Then I should see "Invalid credentials"
  And I should not see "Dashboard"
  And I should see "Sign In"

Scenario: Non-existent user
  Given I am on homepage
  When I go to "/login"
  And I fill in "Email" with "nobody@example.com"
  And I fill in "Password" with "AnyPassword"
  And I press "Sign In"
  And wait for AJAX to finish
  Then I should see "Invalid credentials"
  # Security: same message whether email exists or not
  And I should not see "User not found"

Scenario: Empty credentials
  Given I am on homepage
  When I go to "/login"
  And I press "Sign In"
  Then I should see "Email is required"
  And I should see "Password is required"

Scenario: SQL injection attempt is blocked
  Given I am on homepage
  When I go to "/login"
  And I fill in "Email" with "admin' OR '1'='1"
  And I fill in "Password" with "anything"
  And I press "Sign In"
  And wait for AJAX to finish
  Then I should see "Invalid credentials"
  And I should not see "Dashboard"
```

How It Works:

Security-sensitive applications should:

1. Show the same error message for wrong password AND non-existent user (prevents email enumeration)
2. Never reveal whether an email exists in the system
3. Handle injection attempts gracefully

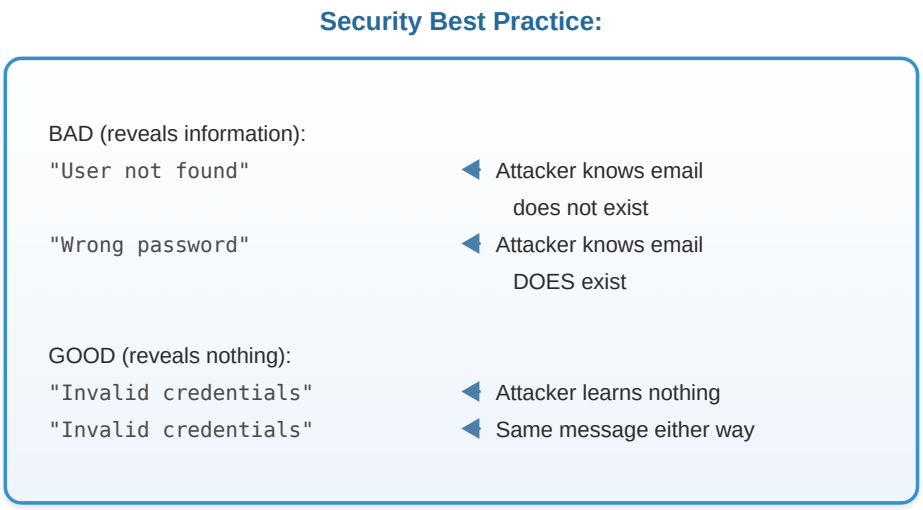


FIGURE 10.3 *Testing Login Failures (Recipe 104)*

Look closely at the non-existent-user scenario: it asserts the generic message *and* asserts `I should not see "User not found"`. The pair is what encodes the security requirement, the positive check alone would pass even if the page showed both messages. The empty-credentials scenario also skips `wait for AJAX to finish` on purpose: required-field validation usually fires client-side before any request leaves the browser, so there is nothing to wait for. And the SQL injection scenario is not really testing your test suite's cleverness, it is a cheap regression tripwire. It will never replace a real security audit, but it costs four lines and fails loudly if someone swaps the ORM call for string concatenation.

At scale: Failure-path scenarios are the ones a PMO can actually point to during a security questionnaire, “email enumeration is covered by an automated check on every build” is a sentence auditors like. Tag these `@security` across all your projects so one CI job can produce a portfolio-wide answer, and so the report for any single project shows the security scenarios as their own line instead of buried among UI checks.

Recipe 105 Using the “Logged In” Step

Problem: Do the math on your suite: 120 scenarios, 90 of them need a logged-in user, and each form login costs eight seconds. That is twelve minutes per run spent typing the same email and password, and worse, 90 scenarios that all fail together the day someone restyles the login page, burying the one real regression in a wall of red. Login should be one reusable step that every scenario shares and only one test (Recipe 103) actually exercises through the form.

Solution:

In your project’s custom step definitions, create a reusable login step:

```
Feature: Admin Dashboard
  Scenario: Admin can view dashboard
    Given I am a logged in user with the "admin" user
    When I go to "/admin/dashboard"
    And wait for AJAX to finish
    Then I should see "Admin Dashboard"
    And I should see "System Statistics"

  Scenario: Editor can view content
    Given I am a logged in user with the "editor" user
    When I go to "/admin/content"
    And wait for AJAX to finish
    Then I should see "Content Management"
```

Custom Step Definition:

```
// tests/step-definitions/custom-auth.js
const { Given } = require('@cucumber/cucumber');

// User credentials map
const users = {
  admin: { email: 'admin@example.com', password: 'AdminPass123', path:
'/login' },
  editor: { email: 'editor@example.com', password: 'EditorPass123', path:
'/login' },
  customer: { email: 'customer@example.com', password: 'CustPass123',
path: '/login' },
  newuser: { email: 'newuser@example.com', password: 'NewPass123', path:
'/login' },
};

Given('I am a logged in user with the {string} user', async function
(role) {
  const user = users[role];
  if (!user) throw new Error(`Unknown user role: ${role}`);

  // Navigate to login page
  await this.page.goto(this.launchUrl + user.path);
  await this.page.waitForLoadState('networkidle');

  // Fill in and submit
  await this.page.getByLabel('Email').fill(user.email);
  await this.page.getByLabel('Password').fill(user.password);
  await this.page.getByRole('button', { name: 'Sign In' }).click();
  await this.page.waitForLoadState('networkidle');
});
```

How It Works:

```
Without reusable step:              With reusable step:

Given I am on homepage              Given I am a logged in user
When I go to "/login"                 with the "admin" user
And I fill in "Email"               When I go to "/admin/dashboard"
  with "admin@example.com"          Then I should see "Dashboard"
And I fill in "Password"
  with "AdminPass123"               ← 1 line instead of 7
And I press "Sign In"
And wait for AJAX to finish
When I go to "/admin/dashboard"
Then I should see "Dashboard"
```

This step:

1. Looks up credentials by role name
2. Navigates to the login page
3. Fills in the form
4. Submits the login
5. Waits for the page to settle

Every test that needs authentication starts with this single step. Two design choices matter here. First, scenarios speak in roles (“admin”, “editor”), not credentials, the email and password live in exactly one place, so rotating a test password is a one-line change instead of a find-and-replace across forty feature files. Second, the step throws on an unknown role rather than failing later with a cryptic “element not found”; when a new hire types `"administrator"` instead of `"admin"`, the error tells them exactly what went wrong. In a real multi-environment setup, read the credentials map from environment variables rather than hardcoding them, so the same step works against every environment your CI targets.

webship-js 2.0 also ships auth-state steps (in auth.steps.js) that save and restore the browser’s storageState, cookies plus localStorage. The pattern: log in through the form once, save the state, and have every later scenario restore it instead of logging in again. It is the same idea as the cookie injection below, but using Playwright’s own persistence machinery and no custom code.

Advanced: Cookie-Based Login (Faster)

For even faster tests, you can inject an authentication cookie directly:

```
Given('I am a logged in user with the {string} user (fast)', async
function (role) {
  const user = users[role];
  // Make API call to get session token
  const response = await fetch(this.launchUrl + '/api/login', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ email: user.email, password: user.password }),
  });
  const data = await response.json();

  // Inject session cookie directly
  await this.context.addCookies([{
    name: 'session_id',
    value: data.sessionId,
    domain: new URL(this.launchUrl).hostname,
    path: '/',
  }]);
});
```

This skips the form entirely and injects the cookie directly. Use this approach when you have many tests that need authentication and the login form itself is not what you are testing. The cookie goes through `this.context.addCookies()`, which sets it at the browser-context level, every page opened in that context carries it, exactly as if the browser had received a Set-Cookie header.

At scale: This step is the single best candidate for a shared step-definition package across your projects. Every product has a login; every product's login differs slightly; and if each suite implements its own version, you will maintain twenty divergent copies. Ship one step that reads per-project configuration (login path, field labels, credentials from env vars) and your QA team learns it once. The runtime saving compounds too, shaving six seconds off ninety scenarios across forty nightly suites is hours of CI time per night, which your DevOps team will notice on the invoice.

Recipe 106 Testing Logout

Problem: A shared kiosk in a clinic, a borrowed laptop in a classroom: a user clicks Logout, sees the login page, and walks away, but the session is still alive on the server, and the next person who presses Back is inside their account. “The login page appeared” and “the session ended” are two different facts, and most logout tests only check the first one. The second is the one that ends up in an incident report.

Solution:

```
Feature: User Logout
  Scenario: Logout ends session
    Given I am a logged in user with the "admin" user
    When I go to "/admin/dashboard"
    Then I should see "Dashboard"

    When I click "Logout"
    And wait for AJAX to finish
    Then I should see "Sign In"
    And I should not see "Dashboard"

  Scenario: Cannot access protected pages after logout
    Given I am a logged in user with the "admin" user
    When I go to "/admin/dashboard"
    Then I should see "Dashboard"

    When I click "Logout"
    And wait for AJAX to finish
    Then I should see "Sign In"

    # Try to access protected page
    When I go to "/admin/dashboard"
    And wait for AJAX to finish
    Then I should see "Sign In"
    And I should not see "Dashboard"

  Scenario: Logout from any page
    Given I am a logged in user with the "admin" user
    When I go to "/admin/users"
    And wait for AJAX to finish
    When I click "Logout"
    And wait for AJAX to finish
    Then I should see "Sign In"
```

How It Works:

FIGURE 10.4 *Testing Logout (Recipe 106)*

The critical assertion after logout is navigating back to a protected page and verifying you are redirected to login. Just checking that the login page appears after clicking "Logout" is not sufficient, the redirect after logout could work while the session is still alive on the server. The second scenario's `When I go to "/admin/dashboard"` after logout is the step that actually proves the session died: the browser still sends whatever cookies it holds, and only a server that has destroyed the session will bounce the request back to login.

If you want belt-and-braces evidence, webship-js's cookie steps can assert the cookie state directly, `Then a cookie with the name "session" should not exist` (the steps read the live cookie jar through the browser context). That distinguishes "cookie cleared but server session alive" from "both properly ended" when you are debugging a logout bug rather than just guarding against one.

At scale: The third scenario, logout from an arbitrary page, earns its place because logout links live in shared layout components, and layout components are exactly what get swapped during redesigns. When a portfolio of sites shares a design system, one broken logout component breaks logout everywhere at once; a per-project scenario that costs five seconds is cheap insurance against a portfolio-wide security regression.

Recipe 107 Testing Password Reset Flow

Problem: Password reset is the feature nobody on the team has used since it was built, everyone logs in with saved credentials. Meanwhile it is the single most-used recovery path your real users have, and the day it silently breaks (a mail template rename, a changed route, a validation regression) your support inbox fills up while every dashboard stays green. You cannot click the emailed link from a browser test, but you can test everything up to the inbox, and that covers most of the ways the flow actually breaks.

Solution:

```
Feature: Password Reset
  As a user who forgot their password
  I want to reset my password
  So that I can regain access to my account

  Scenario: Request password reset
    Given I am on homepage
    When I go to "/login"
    And I click "Forgot Password"
    Then I should see "Reset Password"
    And I should see "Enter your email"
    When I fill in "Email" with "admin@example.com"
    And I press "Send Reset Link"
    And wait for AJAX to finish
    Then I should see "Reset link sent"
    And I should see "Check your email"

  Scenario: Reset with non-existent email
    Given I am on homepage
    When I go to "/login"
    And I click "Forgot Password"
    And I fill in "Email" with "nobody@example.com"
    And I press "Send Reset Link"
    And wait for AJAX to finish
    # Security: same message whether email exists or not
    Then I should see "Reset link sent"

  Scenario: Reset with invalid email format
    Given I am on homepage
    When I go to "/login"
    And I click "Forgot Password"
    And I fill in "Email" with "not-an-email"
    And I press "Send Reset Link"
    Then I should see "Please enter a valid email"
```

How It Works:

The password reset flow typically involves email, so you cannot test the entire flow end-to-end in a browser test. What you CAN test:

- The reset request form works
- Validation on the email field
- Success/error messages
- The security requirement (same message for existent and non-existent emails)

For complete end-to-end reset testing (clicking the link in the email), you would need either:

- A test email server that your test can query via API
- A backdoor endpoint that generates the reset token directly (test environments only)

Note the same enumeration defense from Recipe 104 appears here: the non-existent-email scenario expects "Reset link sent" anyway. A reset form that admits "no account with that email" is an enumeration oracle, and this is the test that keeps the fix fixed. The invalid-format scenario, by contrast, *should* show an error, format validation reveals nothing about which accounts exist.

If you do add a test mail server (MailHog and Mailpit are the usual picks), the rest of the flow becomes testable with the API steps from Chapter 11: the mail server exposes a REST endpoint, so `When I send a GET request to` its messages API retrieves the email, and the reset URL inside it can be visited with a navigation step. That closes the loop without touching a real inbox.

At scale: Decide once, as a team, which tier each project tests to, "form and validation only" or "full loop via test mail server", and record it in the test plan. The worst position is the accidental one: three projects test the full loop, seventeen test nothing past the button, and nobody knows which is which until the day a delivery manager asks "is password recovery covered?" and gets twenty different answers.

Recipe 108 Testing Session Persistence

Problem: A user fills a long form, hits F5 by reflex, and lands on the login page with their work gone. Or your SPA keeps the session in memory instead of a cookie, so every full page load logs everyone out, a bug class that demos never catch because demos never refresh. Sessions are supposed to be boring infrastructure; these scenarios make sure they stay boring across navigation, refresh, and the Back button.

Solution:

```
Feature: Session Persistence
  Scenario: Session survives page navigation
    Given I am a logged in user with the "admin" user
    When I go to "/admin/dashboard"
    Then I should see "Dashboard"

    When I go to "/admin/users"
    And wait for AJAX to finish
    Then I should see "User Management"
    And I should not see "Sign In"

    When I go to "/admin/settings"
    And wait for AJAX to finish
    Then I should see "Settings"
    And I should not see "Sign In"

    When I go to "/admin/dashboard"
    And wait for AJAX to finish
    Then I should see "Dashboard"
    And I should see "Welcome, Admin"

  Scenario: Session survives page refresh
    Given I am a logged in user with the "admin" user
    When I go to "/admin/dashboard"
    Then I should see "Dashboard"

    When I reload the page
```

```
    And wait for AJAX to finish
    Then I should see "Dashboard"
    And I should see "Welcome, Admin"
    And I should not see "Sign In"

  Scenario: Session survives browser back/forward
    Given I am a logged in user with the "admin" user
    When I go to "/admin/dashboard"
    Then I should see "Dashboard"

    When I go to "/admin/users"
    And wait for AJAX to finish
    Then I should see "User Management"

    When I move backward one page
    And wait for AJAX to finish
    Then I should see "Dashboard"
    And I should not see "Sign In"
```

How It Works:

Each scenario exercises a different way the browser re-establishes identity. Navigation between pages sends the session cookie on every request, if any page drops the session, the cookie path or domain is probably scoped too narrowly. `When I reload the page` is the harshest test for SPAs: a full reload wipes all in-memory JavaScript state, so anything that survives it must live in a cookie, localStorage, or the server. And the back/forward scenario catches over-aggressive cache headers that serve a stale “please log in” page from the browser cache.

The repeated `I should not see "Sign In"` after every navigation is the actual assertion; the positive checks just confirm the right page loaded. Sessions rarely fail loudly, they fail by quietly serving the anonymous version of a page, and only the absence check notices.

Pro Tip: If session persistence fails only on one environment, compare cookie attributes before blaming the code. A `Secure` cookie set on an HTTP-only staging environment never comes back; a `SameSite=Strict` cookie disappears on cross-site redirects. webship-js’s cookie assertion steps (`Then a cookie with the name "session" should exist`) let you check what the browser actually holds at each step of the scenario.

Recipe 109 Testing Role-Based Access Control

Problem: A new "Reports" section ships, and three sprints later someone notices that editors, who were never supposed to see financial data, have had a Reports link in their menu the whole time. Nobody broke anything; the developer simply forgot to add the permission check, and no test existed to miss it. Role-based access fails by omission, which means it can only be caught by tests that walk every role through the same pages and assert what each one must *not* see.

Solution:

```
Feature: Role-Based Access
  As a system administrator
  I want different roles to have different access levels
  So that sensitive operations are protected

  Scenario: Admin sees full admin menu
    Given I am a logged in user with the "admin" user
    When I go to "/admin"
    And wait for AJAX to finish
    Then I should see "Dashboard"
    And I should see "Users"
    And I should see "Content"
    And I should see "Settings"
    And I should see "System Logs"

  Scenario: Editor sees limited admin menu
    Given I am a logged in user with the "editor" user
    When I go to "/admin"
    And wait for AJAX to finish
    Then I should see "Dashboard"
    And I should see "Content"
    And I should not see "Users"
    And I should not see "Settings"
    And I should not see "System Logs"

  Scenario: Viewer has read-only access
    Given I am a logged in user with the "viewer" user
```

```
    When I go to "/admin/content"
    And wait for AJAX to finish
    Then I should see "Content"
    And I should not see "Add New"
    And I should not see "Delete"
    And I should not see "Edit"

  Scenario: Admin can manage users
    Given I am a logged in user with the "admin" user
    When I go to "/admin/users"
    And wait for AJAX to finish
    Then I should see "Add User"
    And I should see "Edit"
    And I should see "Delete"

  Scenario: Editor cannot manage users
    Given I am a logged in user with the "editor" user
    When I go to "/admin/users"
    And wait for AJAX to finish
    Then I should see "Access Denied"
    And I should not see "User Management"
```

How It Works:

Role-based testing follows a pattern: log in as each role, navigate to the same page, and verify different content:

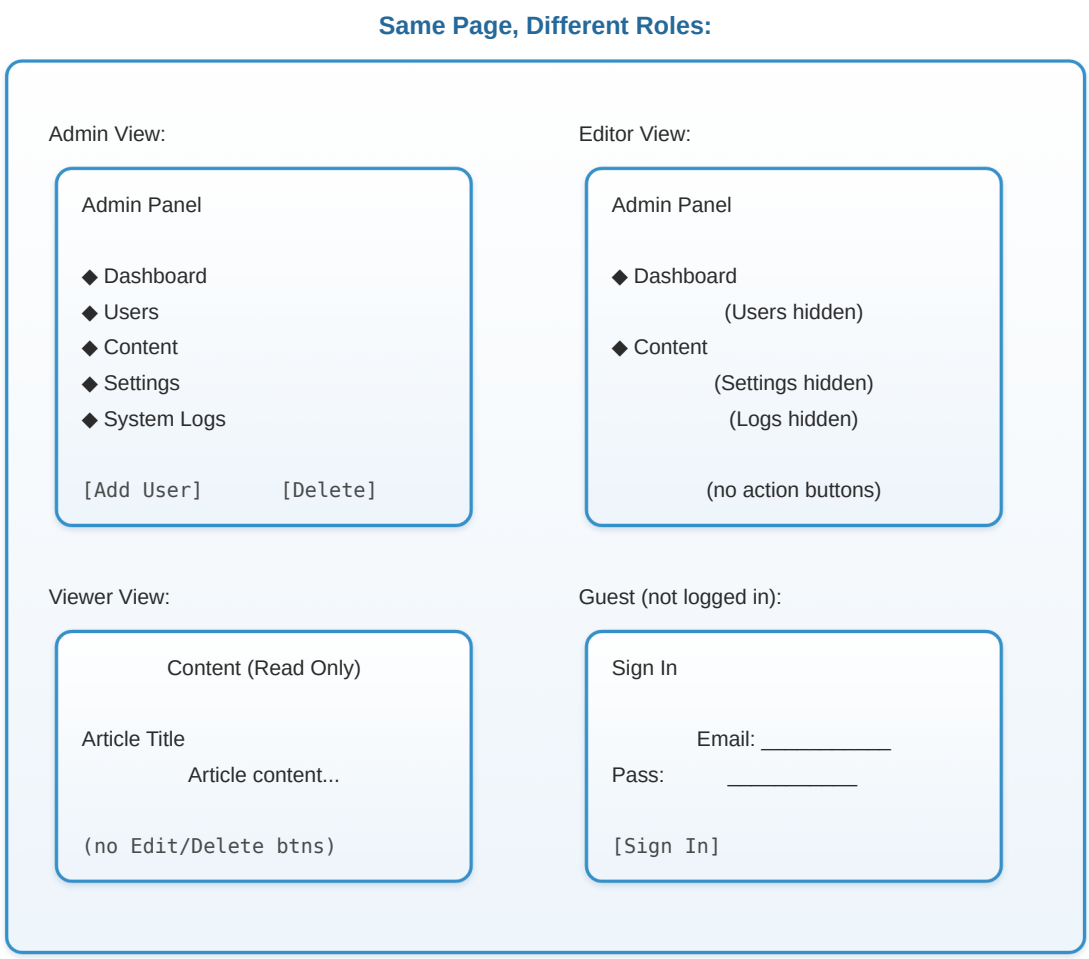


FIGURE 10.5 *Testing Role-Based Access Control (Recipe 109)*

One thing these UI checks deliberately measure: what each role can *see*. Hidden menu items are a usability and information-disclosure concern, and `should not see` covers them well. But a hidden button is not a permission check, the editor who cannot see "Users" in the menu may still be

able to open `/admin/users` directly. That is why the last scenario navigates straight to the URL and expects "Access Denied": it tests the server-side enforcement, not the menu rendering. Always pair the two; plenty of real-world privilege escalations were just unhidden URLs.

Pro Tip: Use `@admin`, `@editor`, `@viewer` tags on your scenarios to organize role-based tests:

```
@admin
Scenario: Admin can manage users
  Given I am a logged in user with the "admin" user
  ...

@editor
Scenario: Editor can manage content
  Given I am a logged in user with the "editor" user
  ...
```

This lets you run all admin tests with `--tags @admin` or all editor tests with `--tags @editor`.

At scale: A role-permission matrix is one of the few test artifacts a product owner can review directly, "should editors see System Logs?" is a product question, not a technical one. Keep the matrix as a document the PO signs off, generate or hand-write the scenarios from it, and treat any diff between the two as a defect in one of them. Across multiple projects, you will also find the roles themselves drift ("editor" means different things on different products); the matrix-per-project forces that conversation early instead of during a security incident.

Recipe 110 Testing Unauthorized Access Redirect

Problem: A new admin route ships in a Friday deploy, `/admin/reports`, added in a hurry, missing the auth middleware that every other route gets. Nothing fails: the page works, the feature works, and anyone with the URL can read it without logging in. Holes like this are invisible to feature tests because feature tests log in first. You need a test that does the opposite: stays anonymous and knocks on every protected door.

Solution:

```
Feature: Unauthorized Access Protection
  Scenario: Unauthenticated user is redirected to login
    Given I am on homepage
    When I go to "/admin/dashboard"
    And wait for AJAX to finish
    Then I should see "Sign In"
    And I should not see "Dashboard"

  Scenario: Unauthenticated user cannot access settings
    Given I am on homepage
    When I go to "/admin/settings"
    And wait for AJAX to finish
    Then I should see "Sign In"

  Scenario: Unauthenticated user cannot access user list
    Given I am on homepage
    When I go to "/admin/users"
    And wait for AJAX to finish
    Then I should see "Sign In"

  Scenario Outline: All admin pages redirect to login
    Given I am on homepage
    When I go to "<path>"
    And wait for AJAX to finish
    Then I should see "Sign In"
    And I should not see "<protected_content>"

    Examples:
      | path              | protected_content   |
      | /admin/dashboard  | Dashboard           |
      | /admin/users      | User Management     |
      | /admin/settings   | Settings            |
      | /admin/content    | Content Management  |
      | /admin/reports    | Reports             |
      | /admin/logs       | System Logs         |

  Scenario: Low-privilege user cannot access admin pages
    Given I am a logged in user with the "viewer" user
```

```
    When I go to "/admin/users"
    And wait for AJAX to finish
    Then I should see "Access Denied"
    And I should not see "User Management"

  Scenario: Editor cannot access system settings
    Given I am a logged in user with the "editor" user
    When I go to "/admin/settings"
    And wait for AJAX to finish
    Then I should see "Access Denied"
    And I should not see "System Settings"
```

How It Works:

The Scenario Outline is particularly powerful here. It generates one test for every admin route, ensuring none of them are accidentally left unprotected:

ROUTE	GUEST	VIEWER	EDITOR
/admin/dashboard	→ Login	✓ View	✓ View
/admin/users	→ Login	✗ Denied	✗ Denied
/admin/content	→ Login	✓ Read only	✓ Full CRUD
/admin/settings	→ Login	✗ Denied	✗ Denied
/admin/reports	→ Login	✓ View	✓ View
/admin/logs	→ Login	✗ Denied	✗ Denied

The anonymous scenarios should start from `Given I am an anonymous user` territory, the step clears all cookies before navigating, which matters in a suite where a previous scenario may have left a logged-in session behind. A “redirect to login” test that accidentally runs with a live admin cookie passes for the wrong reason and guards nothing.

The Examples table doubles as a route inventory. The Friday-deploy hole from the Problem statement is only caught if `/admin/reports` is *in* the table, which makes "add the route to the unauthorized-access outline" a definition-of-done item for any new protected page, the same way developers add the route itself. One line of Gherkin per route is the entire cost.

At scale: This is the recipe to standardize hardest across a portfolio. Keep the protected-route list in a per-project data file next to the suite, reviewed whenever routes change, and have the same outline consume it in every project. When a penetration test or compliance review asks "how do you verify access control?", pointing at this outline, present in all twenty projects, running nightly, with the route list in version control, is a much better answer than describing a manual checklist.

Recipe 111 Testing "Remember Me" Functionality

Problem: "Remember Me" is the kind of checkbox that breaks without anyone filing a bug. Users just notice they keep getting logged out, sigh, and type their password again, you lose a little goodwill every day and never see it in a ticket. Worse is the inverse failure: the box is unchecked but the persistent cookie gets set anyway, which on a shared computer is a real security problem. Both failure modes hide behind a login that otherwise works perfectly.

Solution:

```
Feature: Remember Me
  Scenario: Login without Remember Me
    Given I am on homepage
    When I go to "/login"
    And I fill in "Email" with "admin@example.com"
    And I fill in "Password" with "AdminPass123"
    And I press "Sign In"
    And wait for AJAX to finish
    Then I should see "Dashboard"

  Scenario: Login with Remember Me checked
    Given I am on homepage
    When I go to "/login"
    And I fill in "Email" with "admin@example.com"
    And I fill in "Password" with "AdminPass123"
    And I check "Remember Me"
    And I press "Sign In"
    And wait for AJAX to finish
    Then I should see "Dashboard"

  Scenario: Remember Me keeps user logged in after closing browser
    Given I am on homepage
    When I go to "/login"
    And I fill in "Email" with "admin@example.com"
    And I fill in "Password" with "AdminPass123"
    And I check "Remember Me"
    And I press "Sign In"
    And wait for AJAX to finish
    Then I should see "Dashboard"
    # Note: True browser-close testing requires custom step definitions
    # that create a new browser context while preserving cookies
```

How It Works:

`When I check "Remember Me"` finds the checkbox by its label and checks it; the rest is the normal login flow. What the basic scenarios prove is that the checkbox does not *break* login, a lower bar than it sounds, since a renamed field or a JavaScript handler error on the checkbox can take the whole form down with it.

To test what "Remember Me" actually promises, assert the cookie. A session-only cookie has no expiry and dies with the browser; a remembered session sets a far-future expiry. You cannot read the expiry through a Gherkin step, but you can verify the design that most apps use: a second, separate cookie (often named `remember_token` or similar) that only appears when the box was checked. `Then a cookie with the name "remember_token" should exist` after a checked login, and `should not exist` after an unchecked one, covers both failure modes from the Problem statement, including the security-relevant one, without any custom code.

Note: Testing that "Remember Me" persists across browser sessions requires creating a new browser context while preserving cookies. This is an advanced scenario that goes beyond basic Gherkin steps. For most projects, verifying that the checkbox works, login succeeds with it checked, and the persistent cookie appears only when requested is sufficient.

Recipe 112 Testing OAuth/Social Login

Problem: Someone on the team writes a test that actually logs in through Google, real account, real OAuth dance, and it works beautifully for two weeks. Then Google ships a consent-screen redesign, then the test account trips a bot check and gets locked, then a CAPTCHA appears only on CI's IP range. The test spends the rest of its life flaky or skipped, eroding trust in the whole suite. Third-party login pages are not yours to test; the trick is knowing exactly where your responsibility starts and pointing your automation there.

Solution:

```
Feature: Social Login
  As a user
  I want to log in with my social accounts
  So that I do not need a separate password

  Scenario: Social login buttons are visible
    Given I am on homepage
    When I go to "/login"
    Then I should see "Sign in with Google"
    And I should see "Sign in with GitHub"
    And I should see "Or sign in with email"

  Scenario: Google login button redirects to Google
    Given I am on homepage
    When I go to "/login"
    When I click "Sign in with Google"
    # This will redirect to Google's OAuth page
    # Which we cannot fully test in automated tests
```

How It Works:

OAuth/social login is difficult to test end-to-end in automated tests because:

1. You would need to interact with Google/GitHub/Facebook login pages
2. Those pages may have CAPTCHAs, 2FA, or change their UI
3. Test accounts might get rate-limited or locked

✓ LOGIN BUTTONS EXIST	✗ GOOGLE/GITHUB LOGIN FLOW
✓ Buttons have correct	✗ OAuth callback handling
links/actions	(requires real OAuth)
✓ Post-OAuth landing	✗ Third-party page
page works (if you	interaction
can mock the callback)	
✓ Account linking UI	✗ Token refresh flow

Strategy for testing OAuth:

1. **Unit/integration tests** for the OAuth callback handler (server-side)
2. **UI tests** for button presence and the post-login landing page
3. **Mock the OAuth callback** in test environments to simulate a successful login
4. Test the full OAuth flow manually or in a dedicated E2E environment with test accounts

For option 3, webship-js's network mocking steps do the heavy lifting without any backend changes: `Given the URL "**/oauth/token" returns the JSON:` intercepts the token exchange at the browser level through Playwright's `page.route()`, so you can simulate a successful provider response and test your own callback page, account-creation logic, and post-login landing, the parts that are actually your code. The glob pattern matches the provider's endpoint regardless of host, and the mock lives only inside the scenario that declares it.

At scale: Make "we do not automate third-party login pages" an explicit team policy, written into your testing guidelines, so the debate does not replay on every new project. Each project then carries the same three artifacts: a button-presence scenario, a mocked-callback scenario, and a quarterly manual check of the real flow with a dedicated test account. When the suite is green and a customer still cannot log in with Google, everyone knows immediately that the problem lives in the unmocked third of the flow, which is itself useful triage information.

Recipe 113 Testing Multi-User Workflows

Problem: The editorial workflow works when the developer tests it, as an admin, with one account, end to end. In production, the editor and the approver are different people with different permissions, and the bugs live exactly in the handoff: the submitted draft that never appears in the approver's queue, the rejection reason the editor never sees, the state machine that lets an editor approve their own article. You cannot test a handoff without actually changing hands mid-scenario.

Solution:

```
Feature: Content Approval Workflow
  As a team
  We need content creators and approvers
  So that published content is reviewed

  Scenario: Editor creates content, Admin approves it
    # Step 1: Editor creates a draft
    Given I am a logged in user with the "editor" user
    When I go to "/admin/content"
    And I press "Add Article"
    And wait for AJAX to finish
    And I fill in "Title" with "New Feature Announcement"
    And I fill in "Body" with "We are excited to announce..."
    And I press "Save as Draft"
    And wait for AJAX to finish
    Then I should see "Draft saved"
    And I should see "Status: Draft" in the "New Feature Announcement" row

    # Step 2: Editor submits for review
    When I click "Submit for Review" in the "New Feature Announcement" row
    And wait for AJAX to finish
    Then I should see "Status: Pending Review" in the "New Feature
Announcement" row

    # Step 3: Log out as editor
    When I click "Logout"
    And wait for AJAX to finish
```

```
    # Step 4: Log in as admin and approve
    Given I am a logged in user with the "admin" user
    When I go to "/admin/content"
    And wait for AJAX to finish
    Then I should see "Pending Review" in the "New Feature Announcement"
row
    When I click "Approve" in the "New Feature Announcement" row
    And wait for AJAX to finish
    Then I should see "Status: Published" in the "New Feature
Announcement" row

  Scenario: Admin rejects content
    Given I am a logged in user with the "editor" user
    When I go to "/admin/content"
    And I press "Add Article"
    And wait for AJAX to finish
    And I fill in "Title" with "Incomplete Article"
    And I press "Save as Draft"
    And wait for AJAX to finish
    When I click "Submit for Review" in the "Incomplete Article" row
    And wait for AJAX to finish
    When I click "Logout"
    And wait for AJAX to finish

    Given I am a logged in user with the "admin" user
    When I go to "/admin/content"
    And wait for AJAX to finish
    When I click "Reject" in the "Incomplete Article" row
    And wait for the modal to appear
    And I fill in "Reason" with "Article needs more detail"
    And I click "Submit" in the modal
    And wait for AJAX to finish
    Then I should see "Status: Rejected" in the "Incomplete Article" row
```

How It Works:

Multi-user workflow testing switches between users within a single scenario:

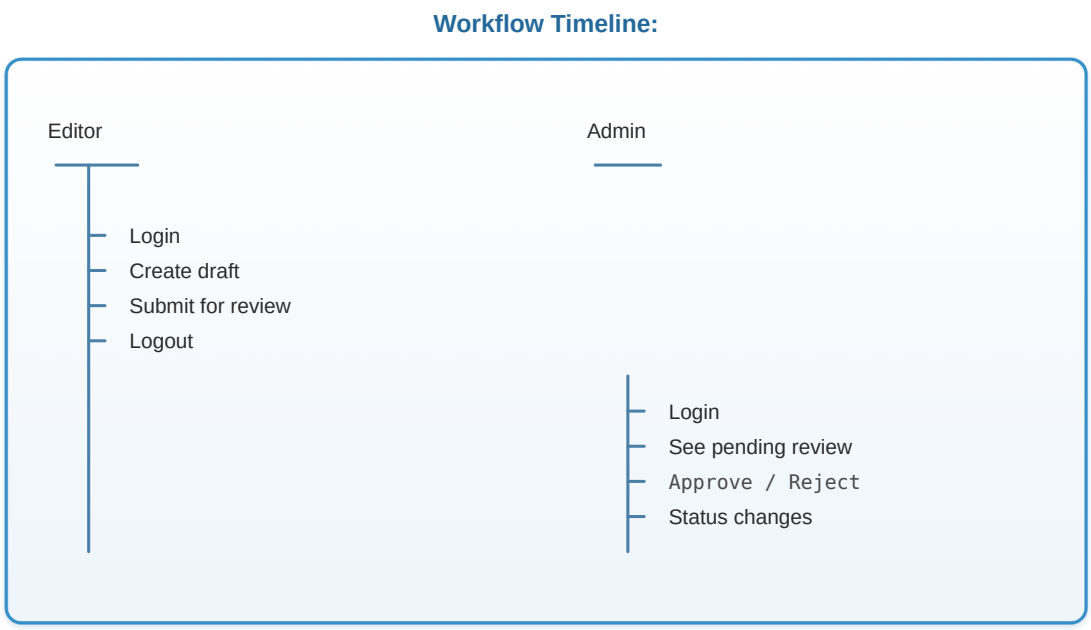


FIGURE 10.6 *Testing Multi-User Workflows (Recipe 113)*

Key points:

- Log out between user switches to start a clean session
- Verify the state changes persist between sessions (editor's submission visible to admin)
- Test both the happy path (approve) and the sad path (reject)

The explicit `When I click "Logout"` before switching users is not ceremony, it is the only thing standing between you and a test that "approves" content while still authenticated as the editor. If your custom login step (Recipe 105) does not clear cookies before logging in, a forgotten logout means the second login silently piggybacks on the first session in some apps. Make the login step defensive: clear cookies first (the same thing `Given I am an anonymous user` does), so a missed logout becomes harmless.

These scenarios are also where the `in the "..." row` steps from Chapter 9 prove their worth outside pure table testing, the article title is the row identifier, and the status assertions (`"Status: Pending Review" in the "New Feature Announcement" row`) track one piece of content through the entire state machine across two user sessions.

At scale: Multi-user scenarios are your longest and most expensive tests, two logins, multiple state transitions, modal interactions. Do not multiply them; pick the one workflow per project that moves money or publishes content, automate that, and cover the permission variations with the cheaper single-role tests of Recipe 109. When a workflow scenario fails on CI, the failure screenshot (captured automatically on failure) plus the step name usually tells you which side of the handoff broke, name your steps so the report reads like the workflow: “editor submits”, “admin approves”.

Recipe 114 Testing User Profile Management

Problem: A user changes their password, the app says “Password changed”, and the new password does not work, because the success message fires before the save that failed. Or the display-name update saves but the old name keeps showing in the header for a week thanks to a cached fragment. Profile pages are low-glamour CRUD that every user touches and no sprint prioritizes, which is exactly the profile of a feature that rots quietly.

Solution:

```
Feature: User Profile
  As a logged in user
  I want to manage my profile
  So that my information is up to date

  Scenario: View profile information
    Given I am a logged in user with the "admin" user
    When I go to "/profile"
    And wait for AJAX to finish
    Then I should see "My Profile"
    And I should see "admin@example.com"
```

```
    And I should see "Admin"

  Scenario: Update display name
    Given I am a logged in user with the "admin" user
    When I go to "/profile/edit"
    And wait for AJAX to finish
    And I fill in "Display Name" with "Tester Demo"
    And I press "Save Profile"
    And wait for AJAX to finish
    Then I should see "Profile updated"
    And I should see "Tester Demo"

  Scenario: Change password
    Given I am a logged in user with the "admin" user
    When I go to "/profile/change-password"
    And wait for AJAX to finish
    And I fill in "Current Password" with "AdminPass123"
    And I fill in "New Password" with "NewAdminPass456"
    And I fill in "Confirm Password" with "NewAdminPass456"
    And I press "Change Password"
    And wait for AJAX to finish
    Then I should see "Password changed"

  Scenario: Password mismatch validation
    Given I am a logged in user with the "admin" user
    When I go to "/profile/change-password"
    And wait for AJAX to finish
    And I fill in "Current Password" with "AdminPass123"
    And I fill in "New Password" with "NewPass1"
    And I fill in "Confirm Password" with "NewPass2"
    And I press "Change Password"
    Then I should see "Passwords do not match"
```

How It Works:

The display-name scenario asserts the success message *and* the new name, the second assertion is the one that catches the caching bug from the Problem statement, because it checks the rendered result, not the server's claim. For the full proof, follow the update with `When I reload the page` and assert "Tester Demo" again; a value that survives a reload came from the backend, not from optimistic JavaScript.

The password-change scenario as written trusts the success message. The stronger version logs out and logs back in with the *new* password, that is the only assertion that proves the credential actually changed. One warning if you add it: the scenario mutates shared test data. Every later scenario that logs in as "admin" now needs the new password, so either change it back at the end of the scenario, or give password-change tests their own dedicated user that nothing else depends on. The second option is the one that will not bite you during parallel runs.

At scale: A dedicated `password-change-user` in the seed data, per project, is the kind of convention worth writing into your shared fixtures standard. Suites that mutate the main admin account's password are the source of some of the most confusing cross-scenario failures you will ever debug, the failure appears in whatever scenario happens to run next, not in the one that caused it.

Recipe 115 Testing API Authentication

Problem: Your UI tests prove the web login works, but half your traffic never touches the UI, mobile apps, integrations, and partner systems all hit the API directly with tokens and keys. When the API's 401 handling regresses, no browser test notices; the first report comes from a partner whose integration broke overnight. API auth needs its own scenarios, and they are cheaper than UI tests: no browser, no rendering, just request and response.

Solution:

```
Feature: API Authentication
  Scenario: Basic Authentication
    Given I am authenticating as "admin" with "password123" password
    When I send a GET request to "/api/users"
    Then the API response code should be 200
    And the API response should contain "users"

  Scenario: Bearer Token Authentication
    Given I set header "Authorization" with value "Bearer
eyJhbGciOiJIUzI1NiJ9..."
    When I send a GET request to "/api/users"
    Then the API response code should be 200

  Scenario: API Key Authentication
    Given I set header "X-API-Key" with value "test-api-key-12345"
    When I send a GET request to "/api/users"
    Then the API response code should be 200

  Scenario: Unauthenticated API request fails
    When I send a GET request to "/api/users"
    Then the API response code should be 401
    And the API response should contain "Unauthorized"

  Scenario: Invalid credentials fail
    Given I am authenticating as "admin" with "wrongpassword" password
    When I send a GET request to "/api/users"
    Then the API response code should be 401

  Scenario: Expired token fails
    Given I set header "Authorization" with value "Bearer expired-token"
    When I send a GET request to "/api/users"
    Then the API response code should be 401
    And the API response should contain "Token expired"
```

How It Works:

webship-js API steps use axios for HTTP requests. The authentication steps set headers on the internal axios configuration:

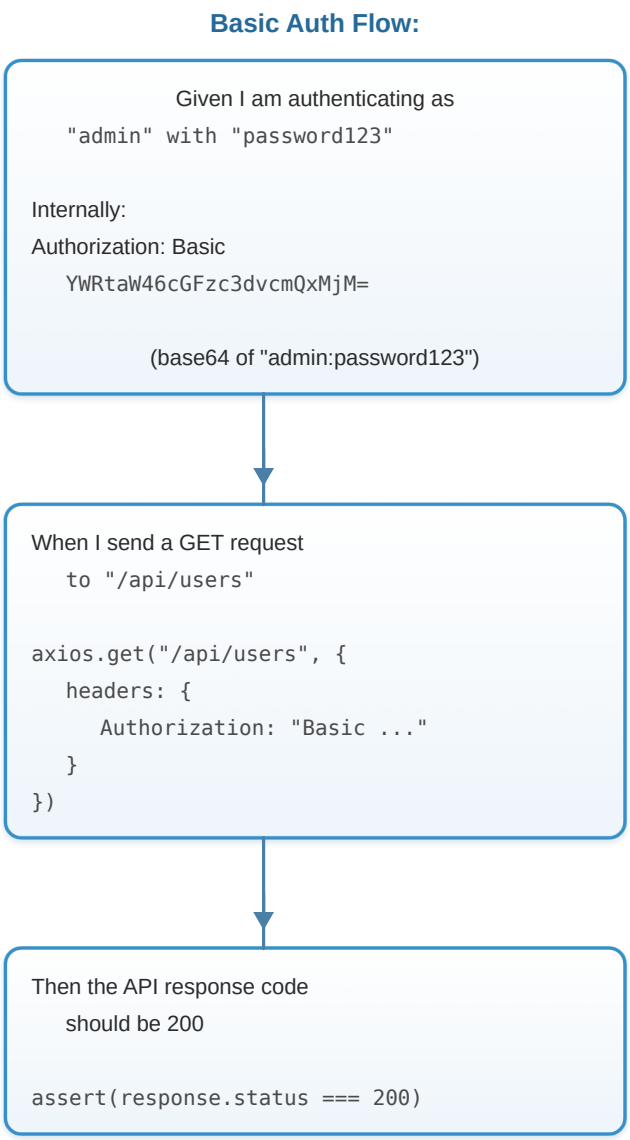


FIGURE 10.7 *Testing API Authentication (Recipe 115)*

Two details worth knowing. First, the headers you set with `Given I set header` accumulate on the axios configuration for the scenario, they ride along on every request that follows, which is exactly what you want for a token but also why a stray header from an earlier step can change a later request's behavior. Each scenario starts clean. Second, these API steps run alongside your browser steps in the same suite, same Cucumber run, same report, but they do not share authentication with the browser. Logging in through the UI does not set an API token, and vice versa; the browser holds cookies, axios holds headers.

The unauthenticated and expired-token scenarios are the ones to never skip. The happy paths get exercised implicitly by every other API test you write; the 401 paths only run when you write them deliberately, and they are where partners and mobile clients feel regressions first.

At scale: Keep tokens and API keys out of feature files in real projects, the docstring values here are illustrations. Wire credentials through CI secrets and a setup step, and tag API auth scenarios (`@api`) so your pipeline can run them as a fast pre-stage before the browser suites: a few seconds of API checks that fail in thirty seconds beat a browser suite that fails after ten minutes on the same broken auth service.

Authentication Testing Strategy

Authentication Testing Priority

Priority 1 (Must Have):
- ✓ Valid login succeeds
- ✓ Invalid login fails with correct message
- ✓ Logout works
- ✓ Protected pages redirect unauthenticated users

Priority 2 (Should Have):
- ✓ Role-based access (admin vs editor vs viewer)
- ✓ Session persists across navigation and refresh
- ✓ Password change works
- ✓ API authentication (Basic Auth, Bearer Token)

Priority 3 (Nice to Have):
- ○ Password reset flow
- ○ Remember Me functionality
- ○ `OAuth/social login button presence`
- ○ Multi-user workflows
- ○ Session timeout testing

Priority 4 (Manual or Integration Tests):
- ○ Full OAuth flow (involves third-party pages)
- ○ Email-based workflows (password reset link)
- ○ Two-factor authentication
- ○ `Rate limiting / account lockout`

FIGURE 10.8 *Authentication Testing Strategy*

Chapter Summary

- Authentication = “Who are you?” / Authorization = “What can you do?”
- Test both successful login AND failure cases
- Security: use the same error message for wrong password and non-existent user
- Create a reusable `Given I am a logged in user with the "X" user` step
- For fast tests, consider cookie injection instead of form-based login
- Always verify that protected pages redirect unauthenticated users
- Test every role against every page/feature to catch missing permission checks
- Multi-user workflows: log out between user switches
- OAuth can only be partially tested in automated UI tests
- API authentication: test Basic Auth, Bearer Token, and API Key patterns
- Password reset: test the request form; full email flow needs a test mail server
- Use Scenario Outline + Examples for comprehensive access control testing
- Tag scenarios by role (`@admin`, `@editor`) for organized test runs

Next: API Testing Recipes, deep-dive into REST API testing with webship-js.

Part III: API and Advanced Testing

Chapter 11: API Testing Recipes

Your web application is an iceberg. The browser-facing UI is the tip, what users see and click. Below the surface sits the API layer: REST endpoints, JSON payloads, authentication headers, status codes, and data contracts. When the UI breaks, users notice. When the API breaks, everything breaks.

webship-js includes a complete API testing toolkit built on axios. You can test REST APIs without writing a single line of JavaScript, every HTTP method, header, authentication scheme, and response assertion is available as a Gherkin step. This means your API tests sit in the same test suite as your UI tests, run with the same command, and speak the same BDD language.

This chapter covers every API testing pattern you need.

Understanding REST API Testing

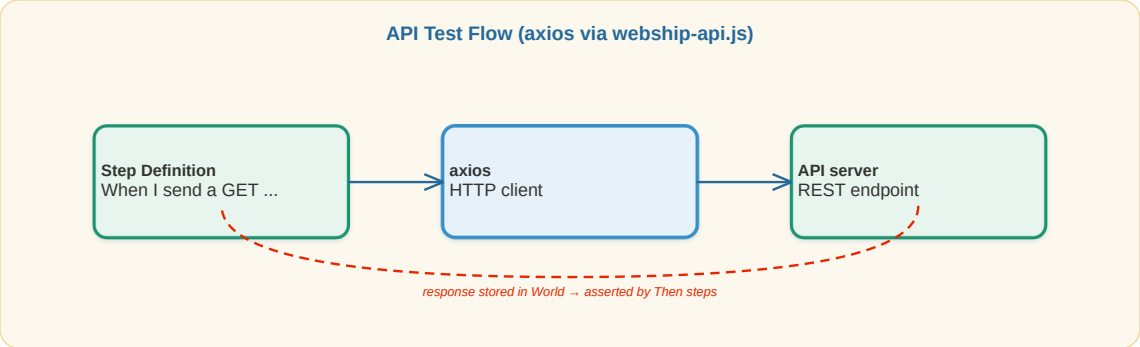


FIGURE 11.1 *Understanding REST API Testing*

webship-js API tests use axios (not the browser) to make HTTP requests directly. This means:

- No browser is needed (faster execution)
- You test the API contract directly (not through the UI)
- You can test endpoints that have no UI at all
- Request/response data is precise and inspectable

UI TEST	API TEST
Browser → UI → API → Database	axios → API → Database
Slow (renders page)	Fast (no rendering)
Tests UI + API together	Tests API in isolation
Catches visual bugs	Catches data/logic bugs
Brittle (UI can change)	Stable (API contract)
"I press 'Save' and see	"POST to /api/users returns
'User created'"	201 with user object"

Recipe 116 Setting Up the API Base URL

Problem: The third time you copy an API suite to a new project, you find yourself doing the same find-and-replace: swap the hardcoded `https://api.client-a.com` for client B's host, in forty scenarios. Miss one and the suite quietly tests the wrong company's API. Base-URL configuration is a one-line step, but where you put that line decides whether your suite is portable or welded to a single environment.

Solution:

```
Feature: API Configuration
  Scenario: Set API base URL
    Given the API base URL is "https://jsonplaceholder.typicode.com"
    When I send a GET request to "/posts/1"
    Then the API response code should be 200

  Scenario: Use relative URL with launch URL
    # If LAUNCH_URL is http://localhost:8080
    Given the base URL is "/api/v1"
    # Requests go to http://localhost:8080/api/v1/...
    When I send a GET request to "/users"
    Then the API response code should be 200

  Scenario: Full URL base
    Given I set the API base URL to "https://api.example.com/v2"
    When I send a GET request to "/products"
    Then the API response code should be 200
```

How It Works:

The base URL step accepts three syntax variants:

- `Given the API base URL is "URL"`
- `Given I set the API base URL to "URL"`
- `Given the base URL is "URL"`

If the URL starts with `http://` or `https://`, it is used as-is. Otherwise, it is appended to the `LAUNCH_URL` environment variable.

BASE URL	ENDPOINT	FINAL URL
https://api.example.com	/users	https://api.
		example.com/
		users
/api/v1	/users	http://local
(LAUNCH_URL=localhost:8080)		host:8080/api
		/v1/users

The relative form is the one that keeps suites portable. `Given the base URL is "/api/v1"` resolves against `LAUNCH_URL`, the same environment variable your browser tests already use (default `http://localhost:8080`). Set `LAUNCH_URL` once per environment in CI, and both your UI scenarios and your API scenarios follow it; nothing in the feature files changes between dev, staging, and the client's UAT server.

At scale: Reserve absolute URLs in feature files for genuinely external services (a public sandbox like jsonplaceholder, a third-party API you integrate with). Everything that belongs to the project under test should go through the relative form. When your DevOps team wires forty pipelines, "every suite obeys LAUNCH_URL" is one convention; "every suite has its own hardcoded hosts in unpredictable places" is forty small archaeology projects.

Recipe 117 GET Requests

Problem: A mobile developer pings you: the app's article screen is blank, but the website shows articles fine. The website goes through its own rendering layer with fallbacks; the app consumes `/posts` raw, and the endpoint started returning an empty array for a specific query parameter three deploys ago. UI tests never saw it because the UI papers over it. GET endpoints need their own assertions, at the contract level, where the mobile app and every other consumer actually live.

Solution:

```
Feature: GET API Requests
  Background:
    Given the API base URL is "https://jsonplaceholder.typicode.com"

  Scenario: Fetch a single resource
    When I send a GET request to "/posts/1"
    Then the API response code should be 200
    And the API response should contain "sunt aut facere"
    And the response should be valid JSON

  Scenario: Fetch a list of resources
    When I send a GET request to "/posts"
    Then the API response code should be 200
    And the response should be valid JSON

  Scenario: Fetch with query parameters
    When I send a GET request to "/posts?userId=1"
    Then the API response code should be 200
    And the API response should contain "userId"

  Scenario: Fetch a non-existent resource
    When I send a GET request to "/posts/99999"
    Then the API response code should be 404
```

How It Works:

`When I send a GET request to "/posts/1"` makes an HTTP GET request using axios:

```
// What webship-js does internally
apiResponse = await axios({
  method: 'GET',
  url: baseURL + '/posts/1',
  headers: apiHeaders,
  validateStatus: () => true  // Don't throw on any status
});
```

The `validateStatus: () => true` is crucial, it means axios will NOT throw an error for 4xx or 5xx status codes. This lets you assert the status code in your Then step instead of catching exceptions. The practical consequence: a 404 is not a test failure unless you say it is. That is what makes the fourth scenario possible, `Then the API response code should be 404` treats the error response as the expected, assertable outcome, which is exactly how you pin down "missing resources return 404, not 200-with-empty-body" as a contract guarantee.

The response object is held for the rest of the scenario, so you can stack as many assertions as you like against one request, status, body text, JSON properties, without re-fetching. One request, many checks, is both faster and kinder to rate-limited APIs.

Recipe 118 POST Requests with JSON Body

Problem: Your team writes API tests in four different styles, one engineer inlines JSON strings, another builds bodies in custom JavaScript steps, a third pastes curl commands into README files as "the tests". Reviewing a pull request means first decoding which dialect it speaks. web-ship-js gives you four built-in ways to send a JSON body; the job is to know what each is for and pick a house style, so the two-hundredth POST test reads like the first.

Solution:

```
Feature: POST API Requests
  Background:
    Given the API base URL is "https://jsonplaceholder.typicode.com"

  Scenario: Create a resource with inline body
    Given I set the request body to '{"title": "New Post", "body":
"Content", "userId": 1}'
    When I send a POST request to "/posts"
    Then the API response code should be 201
    And the API response should contain "New Post"

  Scenario: Create a resource with doc string body
    When I send a POST request to "/posts" with body:
      """
      {
        "title": "My New Post",
        "body": "This is the post content",
        "userId": 1
      }
      """
    Then the API response code should be 201
    And the JSON response should have "title" equal to "My New Post"

  Scenario: Create a resource with data table
    When I send a POST request to "/posts" with values:
      | title  | My New Post              |
      | body   | This is the post content |
      | userId | 1                        |
    Then the API response code should be 201
    And the API response should contain "My New Post"

  Scenario: Create a resource with table-based body setup
    Given I set the request body with:
      | title  | My New Post              |
      | body   | This is the post content |
      | userId | 1                        |
    When I send a POST request to "/posts"
    Then the API response code should be 201
```

How It Works:

webship-js offers four ways to set the POST body:

```
Method 1: Inline JSON string
  Given I set the request body to '{"key": "value"}'

Method 2: Doc String (multiline JSON)
  When I send a POST request to "/url" with body:
    """
    {"key": "value"}
    """

Method 3: Data Table (key-value pairs)
  When I send a POST request to "/url" with values:
    | key | value |

Method 4: Table body setup (separate from request)
  Given I set the request body with:
    | key | value |
  When I send a POST request to "/url"
```

METHOD	BEST FOR
Inline JSON string	Simple, short JSON payloads
Doc String	Complex, nested JSON
Data Table	Flat key-value objects
Table body setup	Reusable body across requests

A nuance worth knowing: the data-table forms build a flat object, every value arrives as a string, and there is no nesting. If the endpoint cares about types (`userId` as a number, `active` as a boolean) or expects nested structures, use the doc string, where you write real JSON and what you

see is exactly what is sent. The doc string is also the form a product owner can read in a pull request and recognize as “the payload from the API docs”, which is worth something during sign-off.

At scale: Pick the doc string as the default house style and write it down. The four variants exist because different payloads have different shapes, but a portfolio where every suite freely mixes all four is a portfolio where nobody can grep for “all tests that create a user”. One style, documented in your team’s testing guideline, makes cross-project review and copy-between-suites cheap.

Recipe 119 PUT Requests (Full Update)

Problem: An integration partner updates a user’s phone number through your API, and the user’s email vanishes. Their developer sent a PUT with only the changed field, your API took “replace the resource” literally, and the missing fields became nulls. Whether that is the partner’s bug or yours depends entirely on what your API contract promises for PUT, and if no test pins that promise down, the answer changes with every refactor.

Solution:

```
Feature: PUT API Requests
  Background:
    Given the API base URL is "https://jsonplaceholder.typicode.com"

  Scenario: Update an entire resource
    When I send a PUT request to "/posts/1" with body:
      """
      {
        "id": 1,
        "title": "Updated Title",
        "body": "Updated body content",
        "userId": 1
      }
      """
    Then the API response code should be 200
    And the JSON response should have "title" equal to "Updated Title"
    And the JSON response should have "body" equal to "Updated body
content"

  Scenario: Update with data table
    When I send a PUT request to "/posts/1" with values:
      | id     | 1                     |
      | title  | Completely New Title  |
      | body   | Completely new body   |
      | userId | 1                     |
    Then the API response code should be 200
    And the API response should contain "Completely New Title"
```

How It Works:

PUT replaces the entire resource. Every field must be included, or the missing fields will be removed (depending on the API design). This is the difference between PUT and PATCH:

```
Original resource:
{ "id": 1, "title": "Old", "body": "Old body", "userId": 1 }

PUT with { "title": "New" }:
{ "title": "New" }                    ← Other fields removed!

PATCH with { "title": "New" }:
{ "id": 1, "title": "New", "body": "Old body", "userId": 1 }
                                        ← Other fields preserved
```

That parenthetical, “depending on the API design”, is exactly what your test exists to pin down. If your API treats PUT as a strict full replacement, send a complete body and assert every field comes back; if it rejects partial PUTs, send one with a missing field and assert the 400 or 422. Either design is defensible. An *untested* design is the problem, because it changes silently with the next framework upgrade, which is how the partner’s phone-number update deleted that email address.

One practical note on the `with values:` table variant: it builds a flat object and every value arrives as a string. For a full-resource PUT that carries numeric ids or booleans, prefer the doc string, `"id": 1` and `"id": "1"` are different payloads, and plenty of APIs accept only one of them.

Recipe 120 PATCH Requests (Partial Update)

Problem: Your mobile team ships autosave: every time a user edits one field, the app PATCHes just that field. Then a backend refactor quietly starts treating every update as a full replacement, and autosaving a new title begins nulling out the body. Each side behaves correctly by its own reading of the API; the data loss lives in the gap between them. PATCH semantics are a contract, and this is the test that gives the contract teeth.

Solution:

```
Feature: PATCH API Requests
  Background:
    Given the API base URL is "https://jsonplaceholder.typicode.com"

  Scenario: Partially update a resource
    When I send a PATCH request to "/posts/1" with body:
      """
      {
        "title": "Only Title Changed"
      }
      """
    Then the API response code should be 200
    And the JSON response should have "title" equal to "Only Title
Changed"
    # Other fields should remain unchanged
    And the JSON response should have property "body"
    And the JSON response should have property "userId"

  Scenario: Update multiple fields
    When I send a PATCH request to "/posts/1" with values:
      | title | Updated Title |
      | body  | Updated Body  |
    Then the API response code should be 200
```

How It Works:

The assertions that matter most are the last two in the first scenario: after patching only the title, the scenario checks that `body` and `userId` still exist in the response. That pair of property checks *is* the PATCH contract, fields you did not send must survive. Drop them and the scenario only proves the title changed, which a destructive full-replace implementation would also pass.

The doc-string form sends exactly the JSON you wrote, so a one-field payload stays a one-field payload. The `with values:` table builds a flat object with string values, fine for titles and bodies, wrong for anything type-sensitive. And as with every API step, the response stays held for the rest of the scenario, so you can stack as many property assertions after one PATCH as the contract deserves.

Recipe 121 DELETE Requests

Problem: A user double-clicks the delete button, the frontend fires two DELETE requests, and the second one returns a 500 that lands in your error tracker every day for a month. Or your nightly cleanup job deletes expired records, hits one that is already gone, and the whole job crashes at 3 a.m. DELETE looks like the simplest verb in HTTP until you ask what happens when the resource is *already deleted*, and that answer is worth pinning down before your consumers discover it for you.

Solution:

```
Feature: DELETE API Requests
  Background:
    Given the API base URL is "https://jsonplaceholder.typicode.com"

  Scenario: Delete a resource
    When I send a DELETE request to "/posts/1"
    Then the API response code should be 200

  Scenario: Delete a non-existent resource
    When I send a DELETE request to "/posts/99999"
    Then the API response code should be 404

  Scenario: Verify deletion
    # Create a resource
    When I send a POST request to "/posts" with body:
      """
      {"title": "Temporary Post", "body": "Will be deleted", "userId": 1}
      """
    Then the API response code should be 201

    # Delete it
    When I send a DELETE request to "/posts/101"
    Then the API response code should be 200
```

How It Works:

The second scenario is the one most suites skip: deleting something that does not exist. A clean 404 lets consumers treat “already gone” as success and retry safely; a 500 turns every retry and every double-click into an incident. Whichever behavior your API has today, this scenario freezes it so a framework upgrade cannot change the answer without a test going red.

The third scenario shows the create-then-delete shape, but be honest about what proves deletion: a 200 from the DELETE only proves the server accepted the request. The full proof is a follow-up GET to the same resource asserting 404, and on a real API you would thread the created resource’s id through with a placeholder (Recipe 129) instead of hardcoding it. Note also that some APIs return 204 No Content for successful deletes rather than 200; both are legitimate designs, so assert whichever yours actually promises.

At scale: Treat DELETE scenarios the way Chapter 9 treats bulk actions: tag them `@destructive` and point them only at environments seeded for the purpose. An API suite that deletes records on a shared staging server can sabotage the browser suite running against the same database five minutes later, a cross-team failure that takes far longer to diagnose than it would have taken to prevent.

Recipe 122 Setting Request Headers

Problem: The request works perfectly in Postman and fails with a 403 in your suite. After an hour of comparing payloads you find it: Postman was quietly sending an API key header you configured months ago and forgot existed. Headers are invisible context, authentication, content negotiation, tenant selection, API versioning all ride in them, and the first time you automate against a gateway that wants three of them at once, you need a clean way to declare headers where a reviewer can actually see them.

Solution:

```
Feature: Request Headers
  Background:
    Given the API base URL is "https://api.example.com"

  Scenario: Set a single header
    Given I set header "Content-Type" with value "application/json"
    And I set header "Accept" with value "application/json"
    When I send a GET request to "/users"
    Then the API response code should be 200

  Scenario: Set header with alternative syntax
    Given I set the header "Authorization" to "Bearer
eyJhbGciOiJIUzI1NiJ9..."
    When I send a GET request to "/protected/resource"
    Then the API response code should be 200

  Scenario: Set multiple headers with data table
    Given I set the following headers:
      | Content-Type  | application/json         |
      | Accept        | application/json         |
      | Authorization | Bearer token123          |
      | X-Custom-Key  | my-custom-value          |
    When I send a GET request to "/users"
    Then the API response code should be 200

  Scenario: API Key authentication via header
    Given I set header "X-API-Key" with value "sk-test-12345"
    When I send a GET request to "/api/data"
    Then the API response code should be 200
```

How It Works:

Headers are stored in a global `apiHeaders` object and sent with every subsequent request in the scenario. Set a header once and it rides along on everything that follows; set the same header again and the new value overwrites the old, which is how you exercise things like token rotation mid-scenario. The `Before` hook resets the whole object between scenarios, so a header set in one scenario cannot contaminate the next, no matter what order the scenarios run in.

SYNTAX	USE CASE
I set header "X" with value "Y"	Single
I set the header "X" to "Y"	Single
the header "X" is "Y"	Single
I set the following headers: (table)	Multiple
I am authenticating as "X" with "Y" password	Basic Auth

At scale: The header values in these examples are illustrations, in a real project, API keys and tokens come from CI secrets, never from feature files committed to the repository. The pattern that holds up across many projects: a `Background` that sets the boring headers every request needs, plus a small custom step that reads the secret from an environment variable and sets the Authorization header. Twenty projects can then share identical feature files while each pipeline injects its own credentials.

Recipe 123 Basic Authentication

Problem: The client's staging server sits behind HTTP Basic Auth, the ops team put it there years ago to keep crawlers and curious eyes out, and every request without credentials bounces with a 401. Your suite, pointed at staging for the first time on a Tuesday afternoon, goes entirely red in thirty seconds. Basic Auth is the oldest gate in the HTTP book and it is still everywhere: internal APIs, staging environments, quick admin endpoints. One step opens it.

Solution:

```
Feature: Basic Authentication
  Background:
    Given the API base URL is "https://api.example.com"
```

```
Scenario: Authenticated request succeeds
  Given I am authenticating as "admin" with "password123" password
  When I send a GET request to "/api/users"
  Then the API response code should be 200
  And the API response should contain "users"

Scenario: Unauthenticated request fails
  When I send a GET request to "/api/users"
  Then the API response code should be 401
  And the API response should contain "Unauthorized"

Scenario: Wrong credentials fail
  Given I am authenticating as "admin" with "wrong-password" password
  When I send a GET request to "/api/users"
  Then the API response code should be 401

Scenario: Authenticated CRUD operations
  Given I am authenticating as "admin" with "password123" password

  # Create
  When I send a POST request to "/api/users" with body:
    """
    {"name": "John", "email": "john@example.com"}
    """
  Then the API response code should be 201

  # Read
  When I send a GET request to "/api/users/1"
  Then the API response code should be 200
  And the API response should contain "John"

  # Update
  When I send a PATCH request to "/api/users/1" with body:
    """
    {"name": "John Updated"}
    """
  Then the API response code should be 200
```

```
# Delete
When I send a DELETE request to "/api/users/1"
Then the API response code should be 200
```

How It Works:

`Given I am authenticating as "admin" with "password123" password` creates a Basic Authentication header:

```
Step: I am authenticating as "admin" with "password123" password

Internally:
  base64("admin:password123") → "YWRtaW46cGFzc3dvcmQxMjM="
  Authorization: Basic YWRtaW46cGFzc3dvcmQxMjM=

This header is sent with every subsequent request in the scenario.
```

Underneath, this is just a header: the step base64-encodes `username:password` and stores the result in the same `apiHeaders` object as Recipe 122, so it follows every request in the scenario and resets before the next one. Two things trip people up. First, base64 is encoding, not encryption, anyone who can read the request can recover the credentials, which is why Basic Auth belongs behind HTTPS only. Second, this header authenticates the axios client and nothing else: your browser steps know nothing about it, because the browser and the API client carry separate identities, the same split Chapter 10 called out.

Recipe 124 Bearer Token Authentication

Problem: Someone pastes a working JWT into the feature file, the suite goes green, everyone moves on. Seven days later the token hits its expiry and every API scenario fails at once, and because the failure says “401” rather than “your hardcoded token died of old age”, the team spends a morning suspecting the API itself. Tokens are born expiring. A suite that depends on a pasted one has a countdown timer built into it; the fix is to earn a fresh token at the start of every scenario.

Solution:

```
Feature: Bearer Token Authentication
  Background:
    Given the API base URL is "https://api.example.com"

  Scenario: Authenticate with Bearer token
    Given I set header "Authorization" with value "Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."
    When I send a GET request to "/api/profile"
    Then the API response code should be 200
    And the API response should contain "name"
    And the API response should contain "email"

  Scenario: Login to get token, then use it
    # Step 1: Login to get a token
    When I send a POST request to "/api/auth/login" with body:
      """
      {"email": "admin@example.com", "password": "AdminPass123"}
      """
    Then the API response code should be 200
    And the JSON response should have property "token"

    # Step 2: Use the token for authenticated requests
    # Note: You would extract the token and set it as a header
    # This requires a custom step definition (see below)

  Scenario: Expired token returns 401
    Given I set header "Authorization" with value "Bearer
expired.token.here"
    When I send a GET request to "/api/profile"
    Then the API response code should be 401
    And the API response should contain "Token expired"

  Scenario: Missing token returns 401
    When I send a GET request to "/api/profile"
    Then the API response code should be 401
    And the API response should contain "Unauthorized"
```

How It Works:

A Bearer token is just an Authorization header with a different prefix, so `Given I set header "Authorization" with value "Bearer ..."` is all the built-in machinery you need, the header lands in the scenario's `apiHeaders` and rides every request that follows. The interesting question is where the token comes from. The login-then-use scenario shows the shape: POST credentials, receive a token, use it. Out of the box the two halves do not connect, because no built-in step copies a value from a response into a header, that is the one gap the custom step below closes.

Custom Step for Token Extraction:

To chain a login request with subsequent authenticated requests, add this custom step:

```
// tests/step-definitions/custom-api-auth.js
const { Then } = require('@cucumber/cucumber');

Then('I save the response {string} as the auth token', function
(propertyPath) {
  const token = getNestedProperty(apiResponse.data, propertyPath);
  if (!token) throw new Error(`No value found at "${propertyPath}" in
response`);
  apiHeaders['Authorization'] = 'Bearer ' + token;
});
```

Then use it:

```
Scenario: Login and use token
  When I send a POST request to "/api/auth/login" with body:
    """
    {"email": "admin@example.com", "password": "AdminPass123"}
    """
  Then the API response code should be 200
  And I save the response "token" as the auth token
  When I send a GET request to "/api/profile"
  Then the API response code should be 200
```

At scale: Put the token-extraction step in your shared step-definition package, with credentials read from environment variables, and every project gets real token auth without pasting anything. One caution from running this across many suites: a fresh login per scenario, multiplied by hun-

dreds of scenarios and dozens of nightly pipelines, is real load on your identity service. If it starts rate-limiting your CI, log in once per run and cache the token, the API twin of the storage-state trick from Chapter 10.

Recipe 125 Form Data Requests

Problem: You point your JSON-speaking test at the legacy login endpoint and get a 400 with an error message that helps nobody. After some archaeology you discover the endpoint predates the JSON era: it wants `application/x-www-form-urlencoded`, the format HTML forms have posted since the nineties. OAuth token endpoints, payment gateways, and anything that began life as a server-rendered form still speak it, and sending JSON at them is answering a letter with a phone call.

Solution:

```
Feature: Form Data Requests
  Background:
    Given the API base URL is "https://api.example.com"

  Scenario: Login with form data
    When I send a POST request to "/login" with form data:
      """
      username=admin
      password=secret123
      remember=true
      """
    Then the API response code should be 200
    And the API response should contain "session"

  Scenario: Submit contact form via API
    When I send a POST request to "/api/contact" with form data:
      """
      name=Tester Demo
      email=rajab@example.com
      message=Hello from the test suite
      """
    Then the API response code should be 200
    And the API response should contain "Message received"
```

How It Works:

The form data step parses `key=value` pairs from the doc string and sends them with `Content-Type: application/x-www-form-urlencoded`:

```
Input:                          Sent as:
username=admin                  Content-Type: application/x-www-form-
urlencoded
password=secret123              Body:
username=admin&password=secret123&remember=true
remember=true
```

The step sets the Content-Type for you, no manual header needed, and URL-encodes the values, so a message containing spaces or ampersands arrives intact. One pair per line keeps the doc string readable in review. When an endpoint rejects your request, let the status code tell you which layer is wrong: a 415 means the content type itself was unexpected, while a 400 or 422 means the format was fine and the values were not, two different bugs that look identical from a distance.

Recipe 126 Response Status Code Assertions

Problem: A framework migration changes the backend's default error handler, and suddenly every failure returns 200 with `{"error": "..."}` tucked in the body. Nothing crashes, it degrades. Clients that branch on status codes stop seeing errors, retry logic stops retrying, and the monitoring dashboard stays green while users stare at broken screens. Status codes are the one part of your contract every consumer relies on without reading the docs, which makes them the cheapest thing to pin down and the most expensive thing to get wrong.

Solution:

```
Feature: Status Code Verification
  Background:
    Given the API base URL is "https://jsonplaceholder.typicode.com"

  Scenario Outline: Common HTTP status codes
    When I send a <method> request to "<endpoint>"
    Then the API response code should be <status>

    Examples:
      | method | endpoint     | status |
      | GET    | /posts/1     | 200    |
      | GET    | /posts/99999 | 404    |
      | POST   | /posts       | 201    |
      | DELETE | /posts/1     | 200    |
```

How It Works:

The Scenario Outline runs once per Examples row, one request, one status assertion, one independent pass/fail line in the report. Because the API steps configure axios with `validateStatus: () => true` (Recipe 117), the 404 row is just another passing test rather than an exception to catch. The real asset is the Examples table itself: method, endpoint, expected status, a status-code contract in a format anyone can extend without touching a step definition. When a new endpoint ships, covering it costs one row.

HTTP Status Code Reference:

CODE	MEANING	WHEN YOU SEE IT
200	**OK**	Successful GET, PUT, PATCH
201	Created	Successful POST (new resource)
204	No Content	Successful DELETE
400	Bad Request	Invalid input/payload
401	Unauthorized	Missing/invalid auth
403	Forbidden	Auth **OK** but no permission
404	Not Found	Resource does not exist
405	Method Not Allowed	Wrong HTTP method for route
409	Conflict	Duplicate resource
422	Unprocessable Entity	Validation failed
429	Too Many Requests	Rate limit exceeded
500	Internal Server Error	Server bug
503	Service Unavailable	Server down/maintenance

At scale: Keep one of these outlines per service and treat its Examples table as the canonical answer to "what should this endpoint return?". It works in both directions: developers add rows when they add routes, and a delivery manager can read the table during release review without anyone translating it. When every project in the portfolio carries the same outline structure, auditing any service's status contract takes thirty seconds, and the question "is release X safe?" starts having evidence behind it.

Recipe 127 Response Body Assertions

Problem: The backend team adopts a new serializer, and every field quietly changes from `snake_case` to `camelCase`. The API still "works", valid JSON, right data, status 200, but every consumer parsing `user_name` now reads `undefined`. A text-level check like "the response contains John" sails straight past this, because the break is in the structure, not the content. Body assertions have to work at the level your consumers actually read: properties, values, and shapes.

Solution:

```
Feature: Response Body Verification
  Background:
    Given the API base URL is "https://jsonplaceholder.typicode.com"

  Scenario: Response contains text
    When I send a GET request to "/posts/1"
    Then the API response should contain "sunt aut facere"
    And the API response should not contain "error"

  Scenario: Response is valid JSON
    When I send a GET request to "/posts/1"
    Then the response should be valid JSON

  Scenario: JSON property exists
    When I send a GET request to "/posts/1"
    Then the JSON response should have property "id"
    And the JSON response should have property "title"
    And the JSON response should have property "body"
```

```
  And the JSON response should have property "userId"

Scenario: JSON property has specific value
  When I send a GET request to "/posts/1"
  Then the JSON response should have "id" equal to 1
  And the JSON response should have "userId" equal to 1

Scenario: JSON property should not exist
  When I send a GET request to "/posts/1"
  Then the JSON response should not have property "password"
  And the JSON response should not have property "secret"

Scenario: Nested JSON property
  When I send a GET request to "/users/1"
  Then the JSON response should have property "address.city"
  And the JSON response should have property "company.name"

Scenario: Response matches expected JSON structure
  When I send a POST request to "/posts" with body:
    """
    {"title": "Test Post", "body": "Test body", "userId": 1}
    """
  Then the API response code should be 201
  And the API response should contain json:
    """
    {
      "title": "Test Post",
      "body": "Test body",
      "userId": 1
    }
    """
```

How It Works:

webship-js provides a rich set of response assertions:

ASSERTION STEP	WHAT IT CHECKS
API response should contain "text"	Text in body
API response should not contain "text"	Text NOT in body
response should be valid JSON	Body is valid JSON
JSON response should have property "x"	Property exists
JSON response should not have property "x"	Property missing
JSON response should have "x" equal to Y	Property value
API response should contain json: """...""""	JSON subset match
response header "X" should be "Y"	Header value
API response code should be N	Status code

The `contain json` assertion does a **subset match**: it checks that every property in the expected JSON exists in the actual response with the same value. The response can have additional properties that are not checked.

```
Subset Match Example:
Expected: { "title": "Test", "userId": 1 }
Actual:   { "id": 101, "title": "Test", "body": "content", "userId": 1 }
Result:   PASS (expected properties match, extra properties ignored)
```

Choose the assertion to match the failure you fear. The `contain "text"` steps do a raw substring search over the serialized body, quick, readable, and blind to structure; they would never notice the snake_case rename as long as the value still appears somewhere. The `JSON response` steps parse the body and walk the property path (dot notation for nesting), so they fail precisely

when a field is renamed, moved, or dropped. And the most underrated line in the whole feature is `should not have property "password"`: positive assertions prove the API does its job, negative ones prove it does not leak, password hashes, internal flags, fields meant for another role.

At scale: Maintain a short "never in a response" list, password, token, secret, internal ids, and assert it as a standard block in every project's API suite. It is three lines of Gherkin per service, it gives the security questionnaire a concrete answer, and the one time it fires (a serializer change that starts including password hashes) it pays for every suite that ever carried it.

Recipe 128 Response Header Assertions

Problem: The platform team swaps the reverse proxy during an infrastructure migration, and the new one drops the `Cache-Control` header your API used to send. Nothing fails, responses just quietly stop being cached, the database absorbs traffic the CDN used to take, and three weeks later someone traces a mysterious load problem back to a header nobody looks at. Headers are where your API meets the infrastructure, and the infrastructure changes without telling your test suite. Unless the suite checks.

Solution:

```
Feature: Response Header Verification
  Background:
    Given the API base URL is "https://jsonplaceholder.typicode.com"

  Scenario: Verify Content-Type header
    When I send a GET request to "/posts/1"
    Then the response header "Content-Type" should contain
"application/json"

  Scenario: Verify Cache-Control header
    When I send a GET request to "/posts/1"
    Then the response header "Cache-Control" should contain "max-age"

  Scenario: Verify custom headers
    Given the API base URL is "https://api.example.com"
    Given I am authenticating as "admin" with "password123" password
    When I send a GET request to "/api/data"
    Then the response header "X-Rate-Limit-Remaining" should contain "99"
```

How It Works:

The step reads the named header from the held response, header names are case-insensitive, per the HTTP spec, and does a substring match on the value. That "contain" semantics is deliberate: asserting `Content-Type` contains `application/json` passes whether or not the server appends `; charset=utf-8`, which is exactly the tolerance you want for a value that varies legitimately between servers.

Header assertions earn their keep on the boundaries you do not own: `Content-Type` for clients with strict parsers, `Cache-Control` for anything sitting behind a CDN, rate-limit headers for partners who budget against them. They are also the assertions that catch infrastructure regressions, the kind that pass every body check, because the application code never changed.

Recipe 129 Placeholder Substitution

Problem: Your scenario creates a user, then needs to fetch, update, and delete that same user, but the id only exists at runtime, inside the create response. So someone hardcodes `/api/users/42`, because user 42 happens to exist on their machine, and the scenario fails in CI where the database starts empty. Hardcoded ids are how API suites end up welded to one particular database snapshot. Placeholders are how you unweld them.

Solution:

```
Feature: Placeholder Substitution
  Background:
    Given the API base URL is "https://api.example.com"
    Given I am authenticating as "admin" with "password123" password

  Scenario: Create and then fetch a resource using placeholder
    Given I set placeholder "{{userId}}" to "42"
    When I send a GET request to "/api/users/{{userId}}"
    Then the API response code should be 200
    And the API response should contain "{{userId}}"

  Scenario: Dynamic placeholders in request body
    Given I set placeholder "{{name}}" to "John Doe"
    Given I set placeholder "{{email}}" to "john@example.com"
    When I send a POST request to "/api/users" with body:
      """
      {
        "name": "{{name}}",
        "email": "{{email}}"
      }
      """
    Then the API response code should be 201
    And the API response should contain "{{name}}"
```

How It Works:

`Given I set placeholder "{{userId}}" to "42"` stores the key-value pair. Before any URL, body, or assertion is processed, all placeholders are replaced:

```
Before:  /api/users/{{userId}}
After:   /api/users/42

Before:  {"name": "{{name}}", "email": "{{email}}"}
After:   {"name": "John Doe", "email": "john@example.com"}
```

Placeholders are reset between scenarios (in the `Before` hook), so they do not leak across tests.

Recipe 130 Complete CRUD API Test

Problem: Each API endpoint passes its own little test, and the API still ships broken. Create returns a 201, sure, but does the thing it created actually come back from a GET? Does an update stick, or does the next read show the old value? Testing each verb in isolation misses the bugs that live in the seams between them, which is exactly where data-integrity problems hide. You need one scenario that walks a resource through its whole life and proves each step left the world in the state the next step assumes.

Solution:

```
Feature: API CRUD Lifecycle
  As an API consumer
  I want to create, read, update, and delete resources
  So that I can manage data through the API

  Background:
    Given the API base URL is "https://jsonplaceholder.typicode.com"

  Scenario: Complete CRUD cycle for posts
    # CREATE
    When I send a POST request to "/posts" with body:
```

```
      """
      {
        "title": "My New Post",
        "body": "This is the content of my new post",
        "userId": 1
      }
      """
    Then the API response code should be 201
    And the response should be valid JSON
    And the JSON response should have property "id"
    And the JSON response should have "title" equal to "My New Post"

    # READ
    When I send a GET request to "/posts/1"
    Then the API response code should be 200
    And the response should be valid JSON
    And the JSON response should have property "title"
    And the JSON response should have property "body"
    And the JSON response should have property "userId"

    # UPDATE (full)
    When I send a PUT request to "/posts/1" with body:
      """
      {
        "id": 1,
        "title": "Updated Post Title",
        "body": "Updated post content",
        "userId": 1
      }
      """
    Then the API response code should be 200
    And the JSON response should have "title" equal to "Updated Post
Title"

    # UPDATE (partial)
    When I send a PATCH request to "/posts/1" with body:
      """
      {
        "title": "Partially Updated Title"
      }
```

```
    """
    Then the API response code should be 200
    And the JSON response should have "title" equal to "Partially Updated
Title"

    # DELETE
    When I send a DELETE request to "/posts/1"
    Then the API response code should be 200
```

How It Works:

This is one scenario, not five, and that is the point. The steps share one `apiResponse` and one base URL across the whole pickle, so the GET genuinely reads back what the POST wrote, and the assertion after the PUT proves the change persisted rather than just that the PUT returned 200. Each `When` overwrites the stored response and each `Then` asserts against that latest one, so the scenario reads top to bottom like the resource's biography. webship-js drives all of it through axios with no browser involved, API scenarios skip Playwright entirely, which is why they run in milliseconds and never flake on a slow render.

The comment markers (`# CREATE`, `# READ`) are free documentation: Cucumber ignores them, but a product owner reading the report sees the lifecycle laid out in plain language. One caveat worth teaching the team, public sandbox APIs like jsonplaceholder fake their writes, so the "updated" value you read back is the one you sent, not one the server stored. Against a real API the same scenario actually proves persistence; against a mock it proves your test wiring.

At scale: A CRUD lifecycle scenario is the most portable test you own. Every resource in every service, users, orders, articles, invoices, has the same create/read/update/delete shape, so this becomes a template your team stamps out per resource across twenty services with only the paths and field names changing. Because it needs no browser, it runs as a near-instant pre-merge gate on every API project at almost no CI cost, catching the data-integrity regressions that are ruinously expensive to find in production.

Recipe 131 Testing Error Responses

Problem: The happy path is the easy half. The expensive bugs are the ones where a bad request gets a 200 and a cheerful "success", a malformed email saved as-is, a duplicate user silently created, garbage JSON swallowed without complaint. Those failures do not announce themselves; they corrupt data quietly and surface as a support ticket three weeks later. You need tests that deliberately send bad input and insist the API pushes back with the right status code and a useful message.

Solution:

```
Feature: API Error Handling
  Background:
    Given the API base URL is "https://api.example.com"

  Scenario: Missing required field
    Given I am authenticating as "admin" with "password123" password
    When I send a POST request to "/api/users" with body:
      """
      {
        "email": "john@example.com"
      }
      """
    Then the API response code should be 422
    And the API response should contain "name is required"

  Scenario: Invalid email format
    Given I am authenticating as "admin" with "password123" password
    When I send a POST request to "/api/users" with body:
      """
      {
        "name": "John",
        "email": "not-an-email"
      }
      """
    Then the API response code should be 422
    And the API response should contain "invalid email"
```

```
Scenario: Duplicate resource
  Given I am authenticating as "admin" with "password123" password
  When I send a POST request to "/api/users" with body:
    """
    {
      "name": "Admin",
      "email": "admin@example.com"
    }
    """
  Then the API response code should be 409
  And the API response should contain "already exists"

Scenario: Invalid JSON payload
  Given I am authenticating as "admin" with "password123" password
  Given I set header "Content-Type" with value "application/json"
  When I send a POST request to "/api/users" with body:
    """
    {not valid json}
    """
  Then the API response code should be 400
  And the API response should contain "Invalid JSON"
```

How It Works:

Each scenario asserts two things together, and you need both. The status code (`422`, `409`, `400`) is the machine-readable contract that clients branch on; the message substring (`"name is required"`, `"already exists"`, `"Invalid JSON"`) is the human-readable part your own front-end shows a user. An API that returns the right code with a useless message, or a friendly message with the wrong code, is half-broken in a way a code-only check would miss. The `should contain` step does a substring match against the whole response body, so it survives small wording changes while still proving the error names the actual problem.

Notice the codes are specific on purpose: `422` for “well-formed but semantically invalid” (the email field is missing), `409` for a conflict (the resource already exists), `400` for “I could not even parse this.” Teaching the team to assert the precise code, not just “something in the 4xx range”, is what stops a future refactor from quietly turning a 422 into a 400 and breaking every client that branched on it.

At scale: Error-path coverage is where API quality actually lives, and it is the first thing dropped under deadline pressure. Make a standing rule that every endpoint ships with at least one negative scenario per documented failure mode, and the rule travels across services because the shapes repeat, missing field, bad format, duplicate, unparseable body are the same four tests everywhere. For a PMO, the count of negative scenarios per endpoint is a concrete, comparable quality metric across the whole portfolio, far more telling than a raw “tests passing” number that the happy path alone can inflate.

Recipe 132 Testing API Pagination

Problem: Pagination works fine with the ten rows in your dev database, then falls over the day a real customer has eight thousand. The off-by-one that drops the last record, the page-beyond-the-end that 500s instead of returning empty, the total-count header that lies, none of these show up until the dataset is big, and by then it is production data and a customer is staring at a missing order. You need to pin down the boundary behavior while the cost of being wrong is still just a red test.

Solution:

```
Feature: API Pagination
  Background:
    Given the API base URL is "https://api.example.com"
    Given I am authenticating as "admin" with "password123" password

  Scenario: First page of results
    When I send a GET request to "/api/users?page=1&per_page=10"
    Then the API response code should be 200
    And the response should be valid JSON
    And the response header "X-Total-Count" should contain "50"

  Scenario: Second page of results
    When I send a GET request to "/api/users?page=2&per_page=10"
    Then the API response code should be 200
    And the response should be valid JSON

  Scenario: Last page with fewer results
    When I send a GET request to "/api/users?page=5&per_page=10"
    Then the API response code should be 200

  Scenario: Page beyond available data
    When I send a GET request to "/api/users?page=100&per_page=10"
    Then the API response code should be 200
    # Should return empty array or 404 depending on API design

  Scenario: Custom page size
    When I send a GET request to "/api/users?page=1&per_page=25"
    Then the API response code should be 200
```

How It Works:

The interesting scenarios here are the boundaries, not the first page. “Page beyond available data” pins down a design decision every paginated API has to make, empty array or 404, and freezes it so a future change cannot silently flip the contract and break clients that depend on one or the other. The `X-Total-Count` assertion uses webship-js’s response-header step to check the count the UI relies on to draw “Page 1 of 5”; a wrong total there means a correct dataset behind a broken pager. Query strings ride along in the request URL exactly as written, so you are exercising the same parameter parsing a real client hits.

These run as fast as any other API scenario because there is no browser, you can cover first page, last page, past-the-end, and custom size in the time a single UI test takes to open Chromium. The honest gap: against a mock API the total count and page math are whatever the mock returns, so meaningful pagination tests need either a real environment or a stub seeded with a known number of rows (Recipe in Chapter 39 covers mocking a fixed dataset for exactly this).

At scale: Pagination, sorting, and filtering are the parameter-driven features most likely to be copy-pasted between services and least likely to be re-tested after the copy. A shared set of boundary scenarios, first, last, past-the-end, custom size, becomes a checklist you apply to every list endpoint across the portfolio, so the off-by-one that one team fixes does not silently reappear in the next service that cloned the code. Pin the empty-vs-404 decision once as an organization-wide API convention and assert it everywhere, so every client team can rely on the same answer.

Recipe 133 Debugging API Responses

Problem: A scenario asserts the response contains `"success"` and it does not, and the failure message tells you what was expected but not what actually came back. You are now guessing: did the request hit the wrong path, did auth fail, did the API return an error envelope you did not anticipate? Guessing at an invisible response is the slowest way to debug an API test. You want to see the real bytes the server sent.

Solution:

```
Feature: API Debugging
  Scenario: Print response for debugging
    Given the API base URL is "https://jsonplaceholder.typicode.com"
    When I send a GET request to "/posts/1"
    Then print API response
    # This will output to the console:
    # GET https://jsonplaceholder.typicode.com/posts/1 => 200:
    # {
    #   "userId": 1,
    #   "id": 1,
    #   "title": "sunt aut facere...",
    #   "body": "quia et suscipit..."
    # }
```

How It Works:

`Then print API response` outputs the request method, URL, status code, and full response body to the console. This is invaluable for debugging:

```
Console output:
GET https://jsonplaceholder.typicode.com/posts/1 => 200:
{
  "userId": 1,
  "id": 1,
  "title": "sunt aut facere repellat provident occaecati...",
  "body": "quia et suscipit\nsuscipit recusandae..."
}
```

Tip: Use `print API response` during development, then remove it before committing. Leaving debug output in your feature files clutters the test output.

At scale: `print API response` is a development aid, not a suite feature, a feature file that ships with it left in is the API equivalent of a forgotten `console.log`, and across twenty repos those add up to noisy CI logs nobody reads. Make “no `print` steps in committed features” a line in the review checklist (it is trivial to grep for in CI: fail the build if a tracked `.feature` contains

`print API response`). When a failure does need a captured response in the report rather than the console, lean on the screenshot-and-artifact path from the reporting recipes instead, so the evidence lands in the shared report where a teammate on another project can read it.

API State Reset Between Scenarios

One critical detail: webship-js resets ALL API state before each scenario:

```
Before(async function () {
  apiResponse = null;    // Clear previous response
  apiRequestData = null; // Clear request body
  apiHeaders = {};       // Clear all headers
  authorization = '';    // Clear auth
  placeHolders = {};     // Clear placeholders
});
```

This means:

- Headers set in one scenario do NOT carry over to the next
- Authentication must be set up in each scenario (or in a Background)
- Placeholders are scenario-scoped
- No state leaks between tests

Use `Background` to avoid repeating common setup:

```
Feature: User API
  Background:
    Given the API base URL is "https://api.example.com"
    Given I am authenticating as "admin" with "password123" password

  Scenario: List users
    When I send a GET request to "/api/users"
    Then the API response code should be 200

  Scenario: Get single user
    When I send a GET request to "/api/users/1"
    Then the API response code should be 200
```

API Testing Quick Reference

WHAT YOU WANT	STEP TO USE
Set base URL	Given the API base URL is "X"
Basic Auth	Given I am authenticating as
	"user" with "pass" password
Set single header	Given I set header "X" with
	value "Y"
Set multiple headers	Given I set the following
	headers: (table)
Set request body (inline)	Given I set the request body
	to '{"key":"value"}'
Set request body (table)	Given I set the request body
	with: (table)
GET request	When I send a GET request
	to "/path"
POST with JSON doc string	When I send a POST request
	to "/path" with body: """
POST with data table	When I send a POST request
	to "/path" with values:

WHAT YOU WANT	STEP TO USE
POST with form data	When I send a POST request
	to "/path" with form data:
Check status code	Then the API response code
	should be 200
Check body contains text	Then the API response should
	contain "text"
Check body NOT contains	Then the API response should
	not contain "text"
Check valid JSON	Then the response should be
	valid JSON
Check JSON property exists	Then the JSON response should
	have property "x"
Check JSON property value	Then the JSON response should
	have "x" equal to Y
Check JSON subset	Then the API response should
	contain json: """...""""
Check response header	Then the response header "X"
	should contain "Y"

WHAT YOU WANT	STEP TO USE
Set placeholder	Given I set placeholder "X"
	to "Y"
Debug response	Then print API response

Chapter Summary

- webship-js API testing uses axios, not the browser, fast and precise
- Every HTTP method is supported: GET, POST, PUT, PATCH, DELETE
- Four ways to send request bodies: inline JSON, doc string, data table, form data
- Basic Auth, Bearer Token, and custom headers are all supported
- Response assertions cover: status codes, body text, JSON properties, headers
- Nested JSON properties use dot notation: `"user.address.city"`
- `contain json` does subset matching, extra properties are ignored
- Placeholders let you reuse values across steps in a scenario
- API state is completely reset between scenarios
- Use `Background` for common setup (base URL, authentication)
- `print API response` for debugging, remove before committing
- Test the complete CRUD lifecycle: Create → Read → Update → Delete
- Test error cases: invalid data, missing auth, duplicate resources

Next: Multi-Browser Testing, running your tests across Chromium, Firefox, and WebKit.

Chapter 12: Multi-Browser Testing

Your users do not all use the same browser. Some are on Chrome, others on Firefox, some on Safari, and a surprising number on Edge (which runs on Chromium). A feature that works perfectly in Chrome might break in Firefox due to a CSS rendering difference or in Safari due to a missing JavaScript API.

Multi-browser testing catches these compatibility bugs before your users do. And with webship-js and Playwright, running your entire test suite across three browser engines is as simple as changing an environment variable.

Here is the uncomfortable part: most teams discover their Safari bugs from support tickets, because everyone develops in Chrome and the test suite runs in Chrome too. The fix is not heroic effort, it is making the other two engines a routine part of the schedule, which this chapter shows you how to do without touching a single feature file.

Understanding Browser Engines

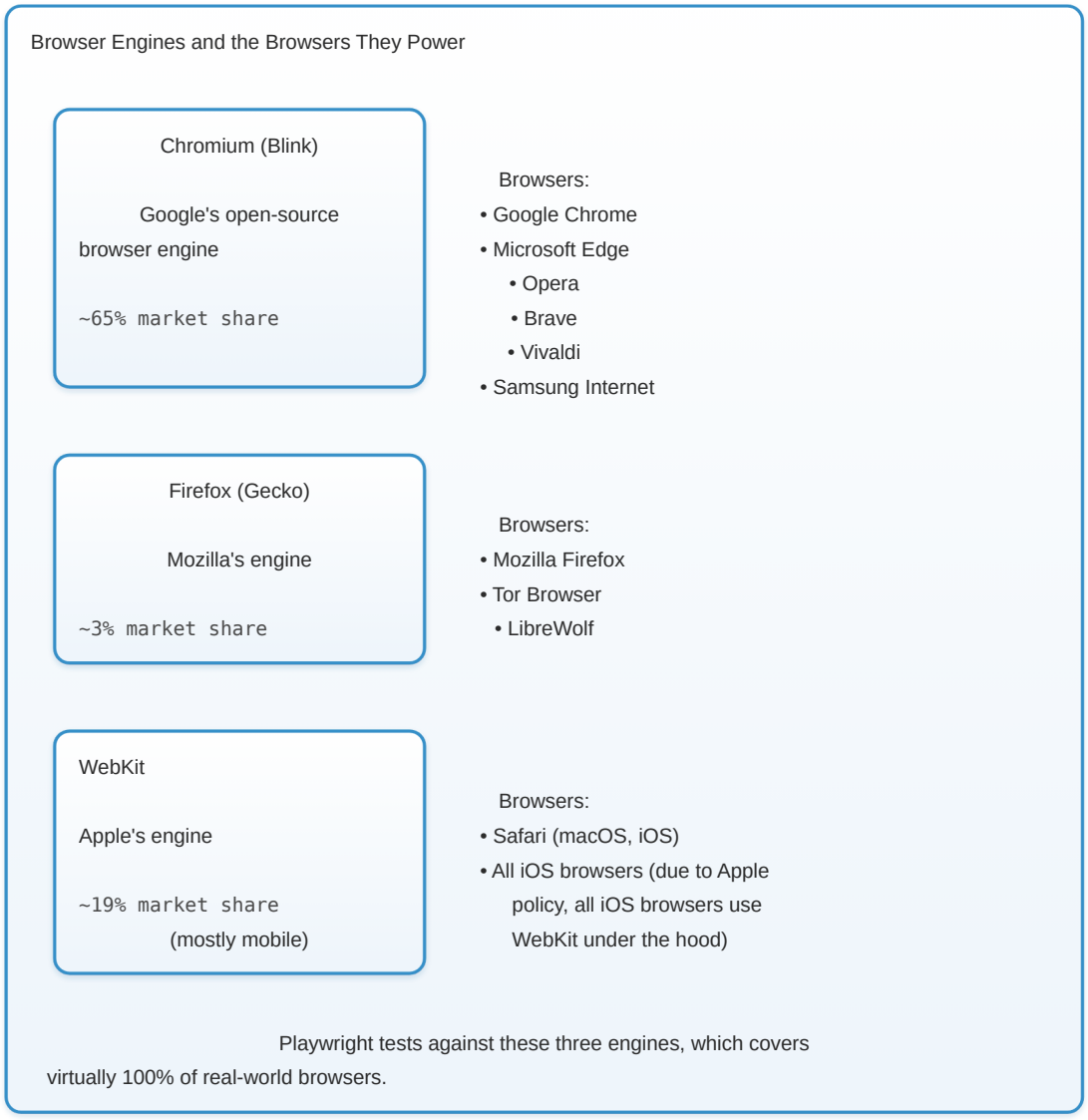


FIGURE 12.1 *Understanding Browser Engines*

By testing against Chromium, Firefox, and WebKit, you cover every major browser engine. This is one of Playwright's key advantages over tools that only support Chromium (like Puppeteer). One engine per family is enough: Chrome, Edge, Opera, and Brave all render with Blink, so a

Chromium pass covers them all. And the market-share numbers above are global averages, the numbers that matter are in your own analytics. A public-facing site can easily see 30% Safari traffic while an internal admin tool sees almost none. Let that data decide how much weight each engine gets in your schedule.

How webship-js Handles Browser Selection

webship-js uses a `BROWSER` environment variable to select which browser engine to use. The configuration is in `playwright.config.ts`:

```
type BrowserName = 'chromium' | 'firefox' | 'webkit';

const browser = (process.env.BROWSER || 'chromium') as BrowserName;
const headless = process.env.HEADLESS !== 'false';

// SLOW_MO default adapts to where the suite runs:
// 0 in CI, 300ms headless on a laptop, 800ms when headed.
const defaultSlowMo = process.env.CI ? 0 : headless ? 300 : 800;

const chromiumArgs: string[] = [
  '--no-sandbox',
  '--disable-dev-shm-usage',
  '--disable-web-security',
  '--ignore-certificate-errors',
  '--incognito',
  '--start-maximized',
];

const config = {
  browser,
  launchOptions: {
    headless,
    slowMo: parseInt(process.env.SLOW_MO || String(defaultSlowMo), 10),
    args: browser === 'chromium' ? chromiumArgs : [],
  },
  contextOptions: {
    viewport: null,
    ignoreHTTPSErrors: true,
  },
};
```

Key points:

- Default browser is `chromium` if `BROWSER` is not set
- Chromium gets special launch arguments (no sandbox, ignore certificate errors, etc.)
- Firefox and WebKit use default arguments
- All browsers respect `HEADLESS` and `SLOW_MO`; the `SLOW_MO` default adapts to context, 0 in CI, 300ms headless, 800ms headed
- `viewport: null` means the page uses the real window size instead of a fixed emulated one
- `ignoreHTTPSErrors: true` works across all browsers

The variable is read once at launch, one process, one engine. That single line of indirection is what keeps multi-browser testing cheap: your features, step definitions, and selectors never branch on the browser, so the cost of adding an engine is CI minutes, not maintenance.

Recipe 134 Testing on Chromium (Default)

Problem: It is your first week wiring tests into a new project, and you need the fastest, most predictable engine to build on. Chromium is where you will write scenarios, debug failures, and gate pull requests, the other engines come later. Before you start trusting green checkmarks, you want to know exactly what the default run does and why it behaves differently inside a container than on your laptop.

Solution:

```
# Default -- no environment variable needed
npx cucumber-js

# Explicit Chromium
BROWSER=chromium npx cucumber-js

# Run specific feature on Chromium
BROWSER=chromium npx cucumber-js tests/features/login.feature
```

```
# This feature runs identically on any browser
Feature: Login on Chromium
  Scenario: Successful login
    Given I am on homepage
    When I go to "/login"
    And I fill in "Email" with "admin@example.com"
    And I fill in "Password" with "AdminPass123"
    And I press "Sign In"
    And wait for AJAX to finish
    Then I should see "Dashboard"
```

Chromium-specific behavior:

```
Chromium Launch Arguments

--no-sandbox                    Runs without sandbox (needed in Docker/CI)
--disable-dev-shm-usage         Prevents crashes on low-memory systems
--disable-web-security          Allows cross-origin requests in tests
--ignore-certificate-errors     Ignores SSL cert issues (dev certs)
--incognito                     Clean browser state for each context
--start-maximized               Full-size viewport

These arguments are only applied to Chromium, not Firefox/WebKit
```

FIGURE 12.2 *Testing on Chromium (Default) (Recipe 134)*

How It Works: When the suite starts, `playwright.config.ts` reads the `BROWSER` environment variable once and falls back to `chromium` when it is unset, so `npx cucumber-js` and `BROWSER=chromium npx cucumber-js` are the exact same run. Chromium is the only engine that gets the extra launch arguments above, and they exist for one reason: Chromium is the engine you will run inside Docker and CI containers, where sandboxing restrictions and tiny `/dev/shm` partitions bite hardest. Everything else about the run is engine-independent. The same built-in

steps execute, and smartSettle still watches `window.__webshipAjaxCount`, pending timers, and DOM mutations before letting each step proceed. Your feature files have no idea which browser they are driving, that is the whole point.

At scale: Make Chromium the pull-request gate on every project, and keep the command identical everywhere: `npx cucumber-js --tags "@smoke"`, no per-project flags. When all twenty suites share the same default, a QA lead can jump between projects without reading each repo's CI file first, and the platform team reasons about one container setup instead of twenty. Save Firefox and WebKit for the nightly schedule, Recipe 137 covers how.

Recipe 135 Testing on Firefox

Problem: A user reports that your date picker shows "Invalid Date", but only in Firefox. Nobody on the team runs Firefox, the suite only runs Chromium, and it turns out the bug has been live for three weeks. You need Firefox in the rotation, so the next Gecko-only regression shows up in a test report instead of a support ticket.

Solution:

```
# Run all tests on Firefox
BROWSER=firefox npx cucumber-js

# Run specific feature on Firefox
BROWSER=firefox npx cucumber-js tests/features/login.feature

# Run tagged tests on Firefox
BROWSER=firefox npx cucumber-js --tags "@smoke"
```

Firefox-specific considerations:

```
Firefox Differences to Watch For

1. CSS Rendering:
                                - Scrollbar styling differs (Firefox uses overlay scrollbars
        on some platforms)
    - Flexbox gap rendering can differ
    - backdrop-filter may not be fully supported

2. JavaScript:
    - Some Web APIs may be behind flags
                                - Date parsing differences (e.g., "2024-01-15" vs "Jan 15")

3. Form Elements:
    - File input styling is different
    - Autofill behavior differs
    - Date/time input widgets look different

4. Network:
    - CORS handling can have subtle differences
    - Cookie SameSite defaults may differ

webship-js handles most of these transparently through
Playwright's unified API.
```

FIGURE 12.3 *Testing on Firefox (Recipe 135)*

How It Works: Setting `BROWSER=firefox` makes Playwright launch its own bundled Firefox build, not whatever copy happens to be installed on the machine, so every developer and every CI runner drives the exact same Gecko version. The Chromium-only launch arguments are skipped automatically; Firefox neither understands nor needs them. Every step behaves identically: `I fill in "Email"` still resolves the field label-first, web-first assertions still poll every 100ms for up to 5 seconds, and smartSettle still waits for AJAX and DOM quiet after every state-changing step. So when a scenario passes on Chromium and fails on Firefox, the difference is almost never the test, it is genuinely your application rendering or parsing something differently, which is exactly the information you wanted.

Pro Tip: When a Firefox failure looks mysterious, rerun just that feature headed: `BROWSER=firefox HEADLESS=false npx cucumber-js tests/features/login.feature`. Watching Gecko render the page usually exposes the difference, a wrapped flexbox item, a differently parsed date, in seconds.

Recipe 136 Testing on WebKit (Safari)

Problem: Your analytics say 19% of visitors are on Safari, and a good chunk of them are on iPhones, where every browser, including "Chrome for iOS", is WebKit underneath. You have no Mac in the CI fleet and nobody clicking through pages on an iPhone before each release. WebKit coverage has quietly become your biggest blind spot.

Solution:

```
# Run all tests on WebKit
BROWSER=webkit npx cucumber-js

# Run specific feature on WebKit
BROWSER=webkit npx cucumber-js tests/features/login.feature
```

WebKit-specific considerations:

```
WebKit (Safari) Differences to Watch For

1. CSS:
    - -webkit- prefixes still needed for some properties
    - position: sticky behavior can differ
    - Smooth scrolling may behave differently

2. JavaScript:
    - Some newer APIs may not be available
    - Regex lookbehind support varies
    - ResizeObserver has had historical bugs

3. Network:
    - Stricter third-party cookie blocking (ITP)
    - More aggressive cache behavior
    - Service Worker support differences

4. Media:
    - Different codec support for video/audio
    - Autoplay policies are stricter

                    Testing on WebKit catches iOS Safari bugs, which is critical
because ALL iOS browsers use WebKit under the hood.
```

FIGURE 12.4 *Testing on WebKit (Safari) (Recipe 136)*

How It Works: Playwright ships a WebKit build that runs on Linux and Windows, not just macOS, that is the trick that makes Safari coverage possible in an ordinary CI container with no Apple hardware in sight. It is the same engine that powers Safari, so the rendering quirks, ITP cookie restrictions, and stricter autoplay rules listed above show up in your test run the way they would on a real device. One practical caveat: on Linux, WebKit needs more system libraries than the other engines, so install it with `npx playwright install --with-deps webkit` (Recipe 142) or run inside the official Playwright Docker image where everything is preinstalled.

At scale: WebKit earns its CI minutes unevenly. A consumer-facing site with heavy mobile traffic should run it nightly on the full suite; an internal back-office tool used on managed Chrome desktops might only need a weekly WebKit smoke run. Decide per project from traffic data, write the decision into the project's CI file, and revisit it quarterly. A blanket "all browsers on everything" policy is how nightly pipelines balloon into three hours of runtime nobody looks at.

Recipe 137 Running Tests Across All Browsers

Problem: The release retro has the same line item two sprints in a row: a bug users found in Firefox, in a flow your Chromium suite covers and passes. The tests were never the problem, the schedule was, because nothing ever ran them on the other two engines. You need one command that sweeps Chromium, Firefox, and WebKit and gives you a per-engine verdict, without maintaining three copies of anything.

Solution:

```
# Run on all browsers sequentially
BROWSER=chromium npx cucumber-js && \
BROWSER=firefox npx cucumber-js && \
BROWSER=webkit npx cucumber-js
```

Create a shell script for convenience:

```
#!/bin/bash
# run-all-browsers.sh

echo "=== Running tests on Chromium ==="
BROWSER=chromium npx cucumber-js "$@"
CHROMIUM_EXIT=$?

echo "=== Running tests on Firefox ==="
BROWSER=firefox npx cucumber-js "$@"
FIREFOX_EXIT=$?

echo "=== Running tests on WebKit ==="
BROWSER=webkit npx cucumber-js "$@"
WEBKIT_EXIT=$?

echo ""
echo "=== Results ==="
echo "Chromium: $([ $CHROMIUM_EXIT -eq 0 ] && echo 'PASS' || echo 'FAIL')"
echo "Firefox:  $([ $FIREFOX_EXIT -eq 0 ] && echo 'PASS' || echo 'FAIL')"
echo "WebKit:   $([ $WEBKIT_EXIT -eq 0 ] && echo 'PASS' || echo 'FAIL')"

# Exit with failure if any browser failed
exit $(( CHROMIUM_EXIT + FIREFOX_EXIT + WEBKIT_EXIT ))
```

npm scripts approach:

```
{
  "scripts": {
    "test": "cucumber-js",
    "test:chromium": "BROWSER=chromium cucumber-js",
    "test:firefox": "BROWSER=firefox cucumber-js",
    "test:webkit": "BROWSER=webkit cucumber-js",
    "test:all-browsers": "npm run test:chromium && npm run test:firefox &&
npm run test:webkit"
  }
}
```

FEATURE	CHROME	FIREFOX	WEBKIT	NOTES
Login	✓	✓	✓	Works everywhere
Forms	✓	✓	✓	Works everywhere
Modals	✓	✓	✗	WebKit animation issue
Tables	✓	✓	✓	Works everywhere
API	✓	✓	✓	No browser dependency
Upload	✓	✗	✓	Firefox file path issue

How It Works: `BROWSER` is read once at launch, so covering three engines means three separate runs, the script simply chains them. Two details make it more than a throwaway loop. First, `"$@"` forwards whatever arguments you pass straight into every run, so `./run-all-browsers.sh --tags "@smoke"` filters all three engines the same way. Second, each exit code is captured separately and summed at the end: the script never stops at the first red engine, so a WebKit failure still leaves you with complete Chromium and Firefox results. That completeness is the point, the matrix above is the real artifact of a multi-browser run, because the failure pattern tells you what kind of bug you have. One engine red means a compatibility bug; all three red means the feature itself is broken. One caveat: every run writes the same JSON report path, so generate or archive the report between runs, `bin/generate-reports.js --browser firefox` stamps the engine into the report metadata so the three results stay distinguishable afterward.

At scale: The sequential script is for laptops. In CI, run the three engines as parallel matrix jobs, wall time stays flat and you pay the same total minutes either way. Across twenty projects, keep the engine list in the shared pipeline template, not in each repo: when WebKit moves from weekly to nightly, that is one template change instead of twenty pull requests. And always pass `--browser` into the report generator, a cross-project dashboard that cannot tell a Firefox run from a Chromium run is just noise.

Recipe 138 Headless vs Headed Mode

Problem: A scenario fails on the third click and the step output tells you nothing you can use, by the time you read it, the browser is gone. You want to watch the run: see the page, the click, the menu that never opened. But the same suite has to run on CI machines that have no display at all, so visibility must be a runtime switch, not a code change.

Solution:

```
# Headless (default) -- no browser window visible
npx cucumber-js

# Headed -- browser window opens, you can watch the test
HEADLESS=false npx cucumber-js

# Headed with slow motion for debugging
HEADLESS=false SLOW_MO=1000 npx cucumber-js

# In CI (always headless)
HEADLESS=true BROWSER=chromium npx cucumber-js
```

How It Works:

HEADLESS MODE (HEADLESS=TRUE)	HEADED MODE (HEADLESS=FALSE)
(Default)	
• No browser window visible	• Browser window opens
• Faster execution	• Slower (rendering visible)
• Uses less memory	• Great for debugging
• Perfect for CI/CD	• Watch tests run in real
• Default in webship-js	time
	• See exactly what the test
	sees
Use for:	Use for:
• CI/CD pipelines	• Writing new tests
• Automated test runs	• Debugging failures
• Quick test execution	• Demos to stakeholders

The switch is the `HEADLESS` environment variable, read by `playwright.config.ts` at launch: the literal string `'false'` opens a real browser window, anything else, including unset, stays headless. Headed mode also changes pace. When you have not set `SLOW_MO` yourself, webship-js picks 800ms between actions for headed runs instead of the headless 300ms, because a browser window you cannot follow is no more useful than no window at all. Everything else, steps, waits, assertions, screenshots, behaves identically in both modes; headless runs the same engine, just without painting to a screen.

Recipe 139 Controlling Browser Speed

Problem: You are demoing the suite to your product owner and the browser flashes through the checkout flow faster than anyone can narrate it. The next morning, the same suite needs every one of those milliseconds back, because CI runs it forty times before lunch. One knob has to serve both audiences, without anyone editing a config file in between.

Solution:

```
# Default adapts to context: 0 in CI, 300ms headless, 800ms headed
npx cucumber-js

# No delay (fastest, for CI)
SLOW_MO=0 npx cucumber-js

# Slow for demonstration
SLOW_MO=1000 npx cucumber-js

# Very slow for debugging
HEADLESS=false SLOW_MO=2000 npx cucumber-js
```

How It Works:

`SLOW_MO` sets the Playwright `slowMo` option, which adds a delay (in milliseconds) before every Playwright action (click, type, navigate). Combined with `HEADLESS=false`, it lets you watch the test run at a pace you can follow.

You rarely need to set it, because the default already adapts to where the suite runs: 0 when `CI` is set (machines do not watch), 300ms for local headless runs, 800ms for headed ones, and an explicit `SLOW_MO` always overrides all three. One thing slow motion is not: a fix for flaky tests. It slows every action equally, which can mask a race condition on your laptop and leave it to resurface in CI at `SLOW_MO=0`. Waiting is smartSettle's job, it holds each step until in-flight requests drain and the DOM goes quiet, so if a scenario only passes with slow motion, treat that as a missing assertion, not a missing delay.

SLOW_MO VALUE	SPEED	USE CASE
0	Fastest	CI/CD pipelines (CI default)
100	Fast	Quick local runs
300	Normal	Default for local headless
500	Moderate	Watching tests run
800	Deliberate	Default for headed runs
1000	Slow	Demos and presentations
2000	Very slow	Step-by-step debugging

Recipe 140 Mobile Browser Emulation

Problem: Analytics says more than half your visitors arrive on phones, yet every test you own runs in a desktop-sized window. The bugs that matter on mobile, a hamburger menu that will not open, a checkout button pushed below the fold by a banner, are invisible at 1920px wide. You do not have a drawer full of test devices, and you do not need one to catch this class of failure.

Solution:

Playwright supports device emulation by setting viewport size and user agent. You can configure this in your project:

```
// playwright.config.ts (custom override)
const devices = require('playwright').devices;

// Example: iPhone 13 emulation
const config = {
  browser: 'webkit',  // Safari engine for iOS
  launchOptions: {
    headless: true,
  },
  contextOptions: {
    ...devices['iPhone 13'],
    ignoreHTTPSErrors: true,
  },
};
```

Testing mobile-specific behavior in Gherkin:

```
Feature: Mobile Experience
  Scenario: Hamburger menu on mobile
    Given I am on homepage
    # On mobile viewport, navigation is collapsed
    Then I should not see "Products"
    When I press "Menu"
    Then I should see "Products"
    And I should see "About"
    And I should see "Contact"
    When I click "Products"
    And wait for AJAX to finish
    Then I should see "Our Products"

  Scenario: Mobile-optimized layout
    Given I am on homepage
    Then I should see "Welcome"
    # Mobile layout shows cards stacked vertically
    And I should see "Featured Products"
```

Device Emulation Profiles:

DEVICE	WIDTH	HEIGHT	SCALE	ENGINE	TOUCH
iPhone SE	375	667	2x	WebKit	Yes
iPhone 13	390	844	3x	WebKit	Yes
iPhone 15 Pro	393	852	3x	WebKit	Yes
iPad Pro 11	834	1194	2x	WebKit	Yes
Pixel 7	412	915	2.6x	Chromium	Yes
Galaxy S23	360	780	3x	Chromium	Yes
Desktop 1080p	1920	1080	1x	Any	No
Desktop 1440p	2560	1440	1x	Any	No
Laptop	1366	768	1x	Any	No

How It Works: Spreading `devices['iPhone 13']` into `contextOptions` swaps webship-js's default `viewport: null` for the full device description Playwright maintains: viewport size, `deviceScaleFactor`, `userAgent`, `isMobile`, and `hasTouch`. Your application genuinely sees a phone, the meta-viewport math, the UA-sniffing code paths, and the touch handlers all engage. Pair the profile with the matching engine: iPhone and iPad profiles belong on `BROWSER=webkit`, because every browser on iOS runs WebKit underneath, while Pixel and Galaxy profiles belong on Chromium. Notice the menu scenario carries no sleep after `I press "Menu"`, webship-js runs its settle check after every state-changing step, waiting for the open animation's DOM mutations to go quiet for 250ms before the next assertion fires, so the menu is actually open when `I should see "Products"` runs. Be honest about what emulation gives you, though: correct geometry, user agent, and touch flags, not a real device's CPU, network, or rendering quirks. That covers layout and behavior bugs, which is most of them. For systematic width-by-width testing against your CSS breakpoints, Chapter 13 picks this up properly.

Recipe 141 Browser-Specific Tags

Problem: WebKit has a known rendering bug in your modal animation, the fix is upstream, months away, and meanwhile every nightly run shows red on a failure nobody is going to act on. A permanently red lane trains the team to stop reading the report, which is worse than having no report. You need a way to park known engine-specific failures without deleting the coverage or losing track of the debt.

Solution:

```
Feature: Cross-Browser Compatibility
  @chromium
  Scenario: Chrome-specific feature
    Given I am on homepage
    # Test something Chrome-specific

  @firefox
  Scenario: Firefox-specific behavior
    Given I am on homepage
    # Test something Firefox-specific

  @skip-webkit
  Scenario: Known WebKit issue
    Given I am on homepage
    # This test fails on WebKit due to a known rendering bug
```

Run with browser-specific tags:

```
# Run only Chromium-specific tests
BROWSER=chromium npx cucumber-js --tags "@chromium"

# Run all tests EXCEPT WebKit-skipped ones
BROWSER=webkit npx cucumber-js --tags "not @skip-webkit"

# Run cross-browser suite (exclude browser-specific tests)
BROWSER=firefox npx cucumber-js --tags "not @chromium and not @webkit"
```

How It Works: There is no magic in these tags, webship-js does not read `@skip-webkit` and skip anything on its own. They are ordinary Cucumber tags, and the pairing happens on the command line: you pick the engine with `BROWSER` and the tag expression with `--tags`, and the convention only holds if your CI files apply both together, every time. That manual link is actually a strength. Tag expressions compose (`not @skip-webkit and @smoke`), and the tag sits in the feature file where a reviewer will see it and ask why. Treat every skip tag as a ledger entry, not a verdict: each one should carry a comment with the issue link, so that `grep -r "@skip-webkit" tests/` returns an open bug list instead of archaeology.

At scale: Agree on one tag vocabulary across all your projects, `@skip-webkit` must mean exactly the same thing in every repo, or the nightly matrix becomes unreadable. The count of skip tags per project is one of the most honest health metrics a QA lead can put on a dashboard: it is the browser debt you have decided to live with, and it should trend toward zero. Review it monthly. A skip tag that survives three reviews without an issue number attached is a test you have silently deleted.

Recipe 142 Installing Browser Binaries

Problem: The suite that ran fine yesterday greets the new CI runner with `browserType.launch: Executable doesn't exist`, and the new hire hits the same wall on their first `npx cucumber-js`, fifteen minutes into day one. Playwright does not drive whatever Chrome happens to be installed on the machine; it drives its own pinned builds, and those have to be put there. Where they come from, where they live, and which system libraries they drag along is worth understanding once, because every machine you ever onboard repeats the same dance.

Solution:

```
# Install all browsers
npx playwright install

# Install specific browser
npx playwright install chromium
npx playwright install firefox
npx playwright install webkit

# Install with system dependencies (Linux CI)
npx playwright install --with-deps chromium
npx playwright install --with-deps firefox
npx playwright install --with-deps webkit
```

CI/CD Installation:

```
# GitHub Actions example
- name: Install Playwright Browsers
  run: npx playwright install --with-deps

# Docker example
FROM mcr.microsoft.com/playwright:v1.58.0-jammy
WORKDIR /app
COPY . .
RUN npm ci
```

How It Works: `npx playwright install` downloads engine builds pinned to the exact Playwright version in your `package.json`, upgrade Playwright and the next install fetches matching browsers, which is how every laptop and runner ends up driving the same build instead of six slightly different ones. On Linux the binaries land in `~/.cache/ms-playwright`, which is what your CI cache step should key on (put the Playwright version in the cache key, or a framework upgrade will restore stale browsers). The `--with-deps` flag additionally installs the system libraries the engines need, a near no-op for Chromium on most images, essential for WebKit, which leans on a long list of GTK and media packages a bare container will not have. The Docker route skips the whole question: the `mcr.microsoft.com/playwright:v1.58.0-jammy` image

ships all three engines and every dependency preinstalled. Just keep the image tag in lockstep with the `playwright` version in `package.json`, a mismatch means re-downloading browsers at container start, which is exactly the slow path you used the image to avoid.

At scale: Browser drift is the platform team's quietest failure mode: a runner built last month and a runner built today carry different engine builds, and a rendering-sensitive test passes on one and fails on the other. Pin one Playwright image tag in the shared pipeline template, upgrade it for all projects at once, and treat that upgrade like any dependency bump, run the nightly matrix on it before rolling it out. Twenty projects on one pinned image is one upgrade to verify; twenty projects each resolving `latest` on their own schedule is twenty surprises.

CI/CD Multi-Browser Strategy

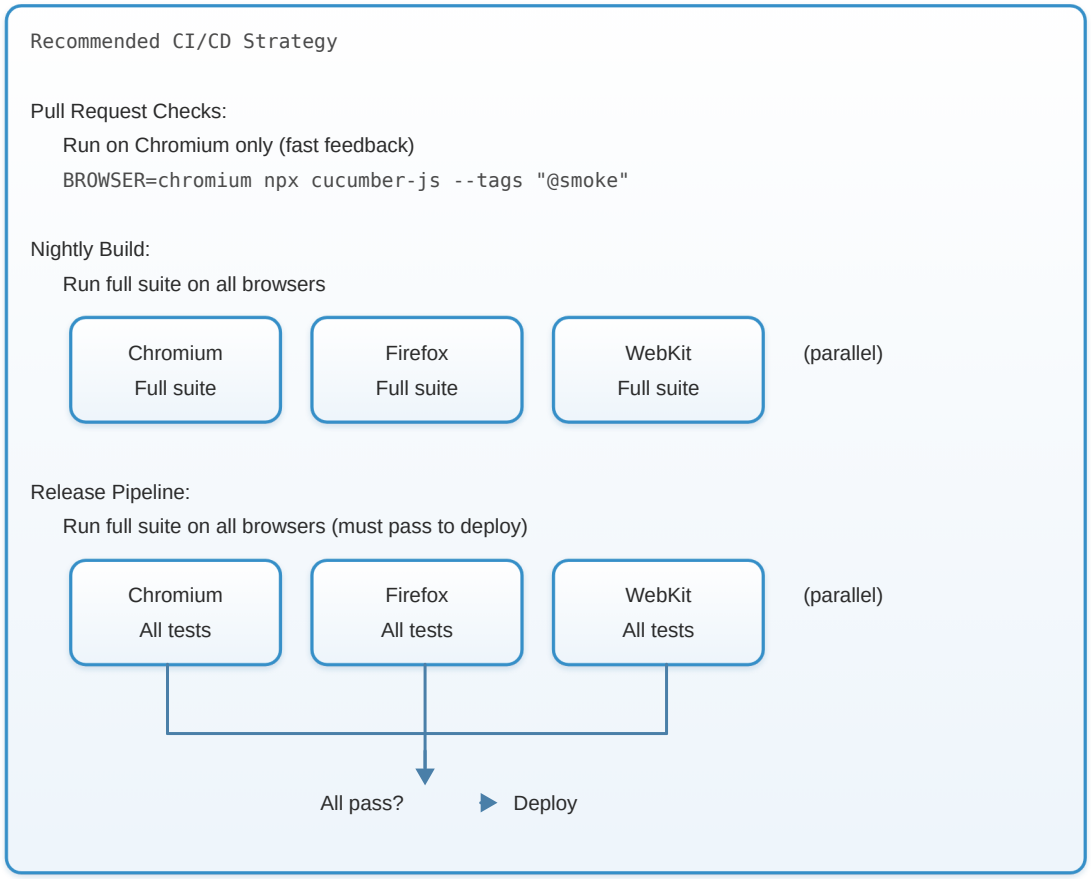


FIGURE 12.5 *CI/CD Multi-Browser Strategy*

This shape is a budget decision as much as a quality one. Running three engines on every pull request would triple feedback time to catch compatibility bugs that almost never hide in a one-line change, so PRs get Chromium speed, and the nightly matrix gets engine breadth while nobody is waiting on it. Wire each nightly lane to stamp its engine into the report metadata (`--browser firefox` on the report generator), and morning triage reads as one matrix instead of three disconnected reports.

Browser Testing Quick Reference

WHAT YOU WANT	COMMAND
Run on Chromium (default)	npx cucumber-js
Run on Firefox	BROWSER=firefox npx cucumber-js
Run on WebKit	BROWSER=webkit npx cucumber-js
Run headed (visible browser)	HEADLESS=false npx cucumber-js
Run with slow motion	SLOW_MO=1000 npx cucumber-js
Run fast (no delay)	SLOW_MO=0 npx cucumber-js
Install browser binaries	npx playwright install
Install with system deps	npx playwright install
	--with-deps
Run all browsers	npm run test:all-browsers
Skip tests for a browser	--tags "not @skip-webkit"

Chapter Summary

- Playwright supports three browser engines: Chromium, Firefox, WebKit
- These three engines cover virtually 100% of real-world browsers
- Switch browsers with `BROWSER=chromium|firefox|webkit` environment variable
- Default is Chromium if `BROWSER` is not set
- Chromium gets special arguments (no-sandbox, ignore cert errors)
- Run headed mode with `HEADLESS=false` for debugging
- Control speed with `SLOW_MO` (0=fastest, 2000=slowest)
- Use browser-specific tags to skip known browser issues
- CI strategy: Chromium for PRs, all browsers for nightly/release
- Mobile emulation uses Playwright's built-in device profiles
- Install browsers with `npx playwright install`
- Your Gherkin features do not change between browsers, the same tests run everywhere

Next: Responsive and Mobile Testing, viewport control, breakpoints, and device emulation.

Chapter 13: Responsive and Mobile Testing

More than half of all web traffic comes from mobile devices. Your application must look and function correctly on screens ranging from a 375px-wide iPhone SE to a 2560px-wide ultrawide monitor. Responsive design is not optional, it is a core requirement.

This chapter shows you how to test responsive behavior, verify mobile layouts, and catch breakpoint-related bugs.

Understanding Responsive Breakpoints

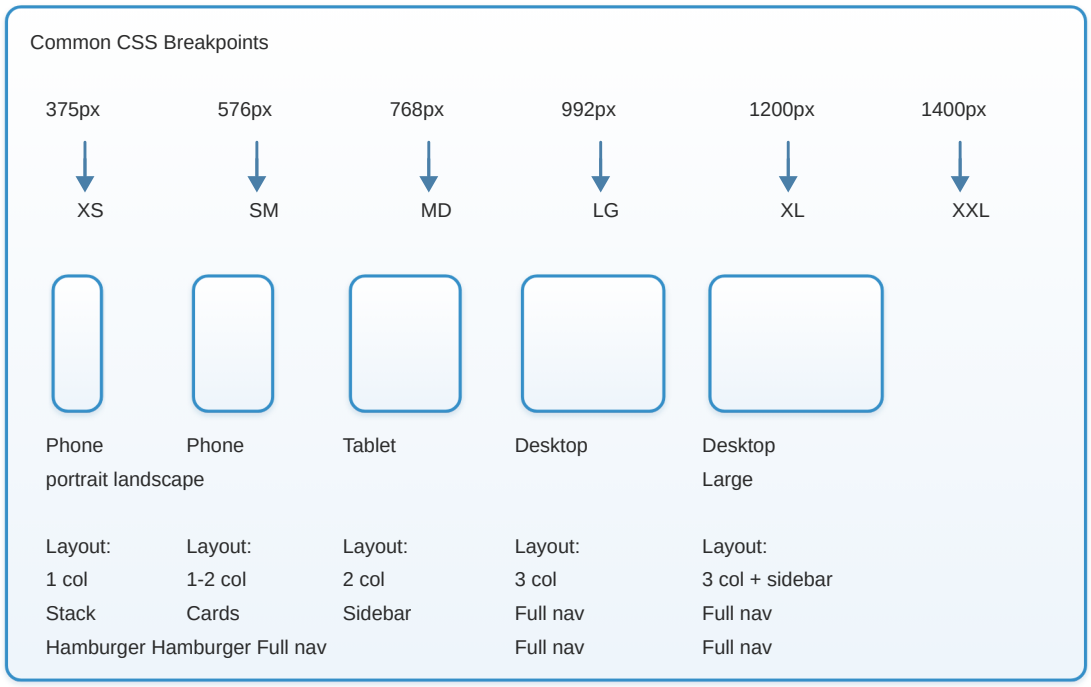


FIGURE 13.1 *Understanding Responsive Breakpoints*

webship-js 2.0 ships this table as a named breakpoints registry in its worldParameters: xs 375x667, sm 576x800, md 768x1024, lg 992x768, xl 1200x900 (the default), xxl 1400x900, and xxxl 1920x1080. You can replace the whole table per project, or per CI run, with the WEBSHIP_SELECTORS_BREAKPOINTS environment variable (JSON). Keep your test viewports aligned with the breakpoints your CSS actually uses; testing at 800px when your CSS switches at 768px proves nothing about either side of the switch.

Recipe 143 Testing at Different Viewport Sizes

Problem: A designer opens the site on her phone and files a bug: the pricing cards overlap at 375px wide. Your suite is green, because every scenario ran at whatever width the CI browser happened to open with. Until you pin viewports in your tests, “responsive design works” is an assumption, not a verified behavior, and the first person to verify it is a user.

Solution:

You can set the viewport through Playwright’s context options. Create custom step definitions for viewport control:

```
// tests/step-definitions/custom-viewport.js
const { Given, When } = require('@cucumber/cucumber');

When('I resize the browser to {int} by {int}', async function (width,
height) {
  await this.page.setViewportSize({ width, height });
});

Given('I am on a mobile device', async function () {
  await this.page.setViewportSize({ width: 375, height: 667 });
});

Given('I am on a tablet', async function () {
  await this.page.setViewportSize({ width: 768, height: 1024 });
});

Given('I am on a desktop', async function () {
  await this.page.setViewportSize({ width: 1920, height: 1080 });
});
```

```
Feature: Responsive Layout
  Scenario: Mobile layout shows hamburger menu
    Given I am on a mobile device
    And I am on homepage
    Then I should not see "Products"
    And I should not see "About Us"
    # Hamburger menu icon should be visible instead
    When I press "Menu"
    Then I should see "Products"
    And I should see "About Us"

  Scenario: Desktop layout shows full navigation
    Given I am on a desktop
    And I am on homepage
    Then I should see "Products"
    And I should see "About Us"
    And I should see "Contact"

  Scenario: Tablet shows adapted layout
    Given I am on a tablet
    And I am on homepage
    Then I should see "Products"
    And I should see "About Us"

  Scenario: Test specific viewport
    When I resize the browser to 1024 by 768
    Given I am on homepage
    Then I should see "Products"
```

How It Works:

`page.setViewportSize()` resizes the rendering surface, re-evaluates every CSS media query, and fires the `resize` event, the same thing that happens when a real user rotates a phone. web-ship-js's playwright.config.ts ships with `viewport: null`, meaning the page follows the launched window size; calling `setViewportSize()` overrides that for the current page. Set it before navigation (as these steps do) so the page loads with its media queries already evaluated at the target width, some layout JavaScript only measures the screen once, on load.

Notice that the mobile and tablet steps match webship-js's own breakpoints registry: 375x667 is xs, 768x1024 is md. Reusing the registry's numbers instead of inventing your own keeps "mobile" meaning the same thing in your custom steps, your selector files, and your CSS. And there is no sleep after pressing "Menu", webship-js runs smartSettle after every state-changing step, waiting for pending timers to drain and the DOM to stay quiet for 250ms, so the menu's open animation finishes before the next assertion fires.

At scale: Do not let each project define its own idea of "mobile". Keep one breakpoints JSON in a shared config package and inject it with WEBSHIP_SELECTORS_BREAKPOINTS in CI. When the design system moves a breakpoint, forty suites move with it, one env var change in the pipeline template instead of forty pull requests, each reviewed by someone who has to re-derive why 768 became 800.

Recipe 144 Testing Mobile Navigation (Hamburger Menu)

Problem: The hamburger menu is the kind of component that breaks quietly. A z-index bump for a promo banner, and the menu now opens behind the hero image; a refactor renames the toggle class and the button does nothing. Nobody on the team notices, because the team lives on desktop, the first report comes from a customer on a phone who cannot reach the Products page at all.

Solution:

```
Feature: Mobile Navigation
  Background:
    Given I am on a mobile device

  Scenario: Hamburger menu opens and closes
    Given I am on homepage
    When I press "Menu"
    Then I should see "Home"
    And I should see "Products"
    And I should see "About"
    And I should see "Contact"

    When I press "Menu"
    Then I should not see "Products"

  Scenario: Navigate via hamburger menu
    Given I am on homepage
    When I press "Menu"
    And I click "Products"
    And wait for AJAX to finish
    Then I should see "Our Products"

  Scenario: Menu closes after navigation
    Given I am on homepage
    When I press "Menu"
    And I click "About"
    And wait for AJAX to finish
    Then I should see "About Us"
    # Menu should auto-close after navigation
    And I should not see "Contact"
```

How It Works:

Menu open and close are usually animated, and animations are where naive suites reach for sleeps. webship-js does not need them. After every state-changing step, an AfterStep hook runs smartSettle(page, 1500), which waits until in-flight fetch/XHR calls hit zero (tracked by window.__webshipAjaxCount), wrapped setTimeout timers drain (window.__webshipPendingTimers), and a MutationObserver reports the DOM quiet for at least 250ms (window.__webshipLastMutation). A 300ms slide-in that toggles classes is exactly the kind of activity those trackers see, so by the time `Then I should see "Home"` runs, the menu is genuinely open. This is why webship-js 2.0 ships no static sleep step at all, a `wait 1s` would only paper over timing the framework already handles.

The third scenario checks the detail teams forget: on single-page apps, navigating from inside the menu often leaves the menu hanging open over the new page. Asserting `I should not see "Contact"` after navigation pins that behavior down before a user finds it.

At scale: Tag these scenarios @mobile and make the tag mean the same thing in every repo. When the CI farm runs nightly across 20 projects, `--tags @mobile` gives one comparable answer to "does mobile navigation work everywhere?", and when a shared navbar component ships a regression, one report tells you which products consumed the broken version, instead of twenty separate bug triages reaching the same conclusion by Friday.

Recipe 145 Testing Responsive Images

Problem: Your team added `srcset` images to cut mobile load time, and the metric moved the wrong way, phones were still downloading the 2400px hero. Nothing functional broke, so no assertion failed; the only symptoms were a slow page and a confused performance review two sprints later. Responsive images fail silently, which makes them exactly the kind of thing worth a test.

Solution:

```
Feature: Responsive Images
  Scenario: Mobile shows optimized images
    Given I am on a mobile device
    When I go to "/products"
    And wait for AJAX to finish
    Then I should see "Products"
    # Verify content loads (images load as part of page)
    And wait for the page to load

  Scenario: Desktop shows full-size images
    Given I am on a desktop
    When I go to "/products"
    And wait for AJAX to finish
    And wait for the page to load
    Then I should see "Products"
```

Note: Testing that the correct image *variant* loaded (small vs large) typically requires checking the rendered image dimensions or the `currentSrc` property. This needs a custom step:

```
// tests/step-definitions/custom-responsive-image.js
const { Then } = require('@cucumber/cucumber');
const assert = require('assert');

Then('the {string} image should be narrower than {int} pixels', async
function (alt, maxWidth) {
  const img = this.page.locator(`img[alt="${alt}"]`).first();
  const box = await img.boundingBox();
  assert(box.width < maxWidth,
    `Image "${alt}" is ${box.width}px wide, expected less than
${maxWidth}px`);
});
```

How It Works:

`boundingBox()` returns the rendered size in CSS pixels after layout, what the user actually gets at this viewport, and a fast proxy for "the right variant loaded". To verify the variant directly, read `currentSrc` (not `src`) inside `page.evaluate()`: `currentSrc` is the URL the browser actually chose from the srcset after applying device pixel ratio and the `sizes` attribute. One caution, assert on dimensions or the chosen URL, never on bytes downloaded. Browsers may legitimately reuse a larger cached variant after a resize, and a byte-level assertion turns that optimization into a flaky failure you will chase for a week.

Recipe 146 Testing Media Query Breakpoints

Problem: A developer "tidies up" a media query from `max-width: 768px` to `max-width: 767.98px`, or flips a `min-width` to a `max-width`, and the layout now has a one-pixel band where the page shows neither the hamburger nor the full menu. iPad users in portrait, who sit at exactly 768px, get the broken state. You will never catch this by testing "mobile" and "desktop"; you catch it by testing the boundaries themselves.

Solution:

```
Feature: Breakpoint Testing
  Scenario Outline: Layout adapts at breakpoints
    When I resize the browser to <width> by 800
    Given I am on homepage
    Then I should see "<expected_element>"

    Examples:
      | width | expected_element    |
      | 375   | Menu                |
      | 576   | Menu                |
      | 768   | Products            |
      | 992   | Products            |
      | 1200  | Products            |

  Scenario: Sidebar appears on large screens
    When I resize the browser to 1200 by 800
    Given I am on homepage
    When I go to "/blog"
    And wait for AJAX to finish
    Then I should see "Recent Posts"
    And I should see "Categories"

  Scenario: Sidebar hidden on small screens
    When I resize the browser to 375 by 667
    Given I am on homepage
    When I go to "/blog"
    And wait for AJAX to finish
    Then I should see "Recent Posts"
    And I should not see "Categories"
```

How It Works:

The Scenario Outline tests the same page at each breakpoint, verifying that the expected elements are visible. This catches CSS bugs where a breakpoint is slightly off (e.g., `max-width: 767px` instead of `max-width: 768px`).

Each Examples row runs as its own scenario with a fresh World and a fresh browser, the resize never leaks between rows, and a failure at 768px shows up as its own line in the report rather than burying itself inside a loop. That is also why the resize step comes before navigation in each row: the page loads with its media queries already evaluated at the target width, exactly as it would on a real tablet. When a row fails, the report hands you the exact width, which usually points straight at the off-by-one in the CSS.

If you want to be even stricter, test one pixel either side of each boundary (767 and 768, 991 and 992). Boundary pairs are where every breakpoint bug lives.

ELEMENT	375PX	768PX	992PX	1200PX
Nav Menu	Hamburger	Full	Full	Full
Sidebar	Hidden	Hidden	Visible	Visible
Grid Cols	1	2	3	4
Footer	Stacked	2 cols	4 cols	4 cols

At scale: Breakpoint matrices multiply fast: 5 widths x 3 browsers x 20 projects is 300 browser launches before you have tested a single behavior, and your DevOps team will notice the nightly bill. Run the full matrix under its own tag (@breakpoints) on the nightly schedule, keep pull-request runs to the default xl viewport plus one mobile width, and stamp the report with `--meta-data Viewport=375` so the cross-project dashboard can slice failures by width instead of someone reading forty reports.

Recipe 147 Testing Touch-Friendly Elements

Problem: Support keeps logging the same complaint from mobile users: “the Save button does not work.” It works fine, it is 28 pixels tall and sits 4 pixels from a Cancel link, so thumbs miss it or hit Cancel instead. Apple’s HIG says 44x44 CSS pixels minimum, Material Design says 48x48dp, and nobody on the team measures either until the complaints arrive.

Solution:

```
// tests/step-definitions/custom-touch-target.js
const { Then } = require('@cucumber/cucumber');
const assert = require('assert');

Then('the {string} button should be at least {int}px tall', async function
(text, minHeight) {
  const button = this.page.getByRole('button', { name: text }).first();
  const box = await button.boundingBox();
  assert(box.height >= minHeight,
    `Button "${text}" is ${box.height}px tall, minimum is
${minHeight}px`);
});
```

```
Feature: Touch-Friendly UI
  Background:
    Given I am on a mobile device

  Scenario: Buttons are large enough to tap
    Given I am on homepage
    When I go to "/login"
    Then the "Sign In" button should be at least 44px tall

  Scenario: Mobile form fields are accessible
    Given I am on homepage
    When I go to "/contact"
    Then I should see "Name"
    And I should see "Email"
    And I should see "Message"
    # Fields should be tappable and fillable on mobile
    When I fill in "Name" with "Rajab"
    And I fill in "Email" with "rajab@example.com"
    And I fill in "Message" with "Mobile message"
    And I press "Send"
    And wait for AJAX to finish
    Then I should see "Message sent"
```

How It Works:

`getByRole('button', { name: text })` resolves the button through the accessibility tree, the same mechanism behind webship-js's built-in `When I click the "Save" button` step, so it matches `<button>`, `input[type=submit]`, and ARIA `role="button"` elements alike. `boundingBox()` then returns the element's rendered size in CSS pixels at the current viewport. That last part is the point of the Background line: a button can be a comfortable 48px on desktop and shrink below the threshold at 375px, where responsive padding collapses. Measure at the viewport where thumbs actually do the tapping.

The second scenario looks redundant, fill a form, like Chapter 6, but running it at mobile width catches a class of bug desktop tests cannot: fields hidden behind a sticky footer, labels truncated into ambiguity, a Send button pushed off-screen by the on-page keyboard layout.

Responsive Testing Quick Reference

WHAT YOU WANT	APPROACH
Test mobile layout	Given I am on a mobile device
Test tablet layout	Given I am on a tablet
Test desktop layout	Given I am on a desktop
Test specific viewport	When I resize the browser to
	WxH
Test hamburger menu	Press "Menu" → verify items
Test breakpoint transitions	Scenario Outline with widths
Test touch targets	Custom step: bounding box
Test mobile device emulation	Playwright devices config

Chapter Summary

- More than half of web traffic is mobile, responsive testing is essential
- Custom viewport steps let you test at any screen size
- Test breakpoint boundaries where layout changes occur
- Mobile navigation (hamburger menus) needs explicit open/close testing
- Responsive images and media queries are testable through content visibility
- Touch target size validation ensures mobile usability
- Scenario Outlines with viewport widths efficiently test all breakpoints
- Playwright’s device profiles simulate real mobile devices
- Combine viewport testing with browser testing for comprehensive coverage

Next: Screenshot and Visual Testing, capturing evidence and detecting visual regressions.

Chapter 14: Screenshot and Visual Testing

Sometimes a test can pass every assertion and still miss a bug. The login button works, but it is hidden behind a misaligned div. The table data is correct, but the columns overlap. The modal opens, but the text is invisible because white text is on a white background.

These are visual bugs, and they slip through functional assertions because `I should see "Login"` checks the DOM, not the pixels. Screenshot testing catches what assertions cannot.

This chapter covers capturing screenshots, organizing them as evidence, and using them to detect visual regressions.

Understanding Visual Testing

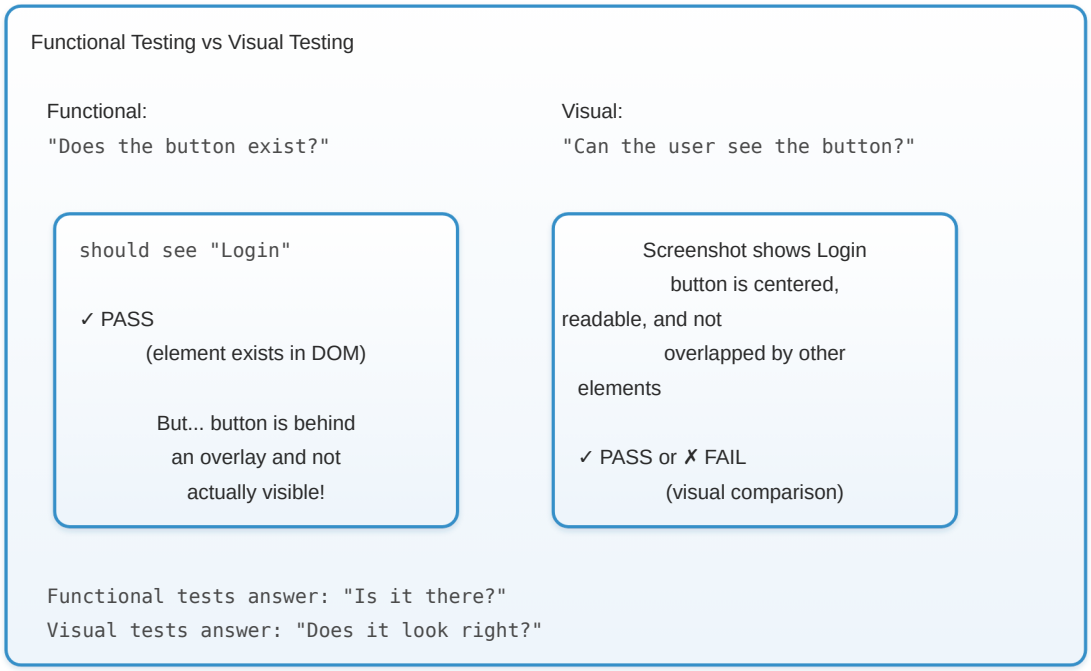


FIGURE 14.1 *Understanding Visual Testing*

Recipe 148 Capturing Screenshots on Test Failure

Problem: The nightly run fails at 2 a.m. on a scenario that passes every time you run it locally. The error says “expected to see Dashboard”, but what was actually on screen? A login error? A maintenance page? A cookie banner covering the button? Without a picture of that moment you are debugging blind, and the usual outcome is the worst one: someone re-runs the job, it goes green, and the real bug ships.

Solution:

Add an After hook that captures a screenshot on failure:

```
// tests/step-definitions/custom-screenshots.js
const { After, Status } = require('@cucumber/cucumber');
const path = require('path');
const fs = require('fs');

After(async function (scenario) {
  if (scenario.result.status === Status.FAILED) {
    // Create screenshots directory if needed
    const screenshotDir = path.join(process.cwd(), 'screenshots');
    if (!fs.existsSync(screenshotDir)) {
      fs.mkdirSync(screenshotDir, { recursive: true });
    }

    // Generate filename from scenario name
    const name = scenario.pickle.name
      .replace(/[^a-zA-Z0-9]/g, '-')
      .replace(/-+/g, '-')
      .toLowerCase();
    const timestamp = new Date().toISOString().replace(/[:.]/g, '-');
    const filename = `${name}-${timestamp}.png`;

    // Capture screenshot
    await this.page.screenshot({
      path: path.join(screenshotDir, filename),
      fullPage: true,
    });

    // Attach screenshot to Cucumber report
    const screenshot = await this.page.screenshot({ fullPage: true });
    this.attach(screenshot, 'image/png');
  }
});
```

```
Feature: Screenshot on Failure
  Scenario: This test will produce a screenshot if it fails
    Given I am on homepage
    When I go to "/login"
    And I fill in "Email" with "admin@example.com"
    And I fill in "Password" with "WrongPassword"
    And I press "Sign In"
    And wait for AJAX to finish
    # If this fails, a screenshot is automatically captured
    Then I should see "Dashboard"
```

How It Works:

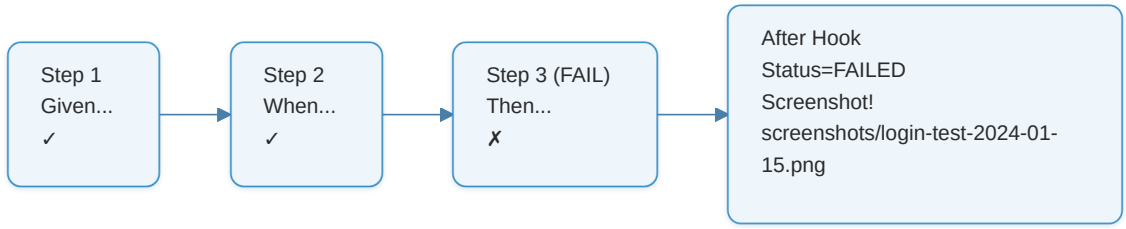


FIGURE 14.2 *Capturing Screenshots on Test Failure (Recipe 148)*

The After hook checks the scenario result status. If it is `FAILED`, it:

1. Creates a `screenshots/` directory if it does not exist
2. Captures a full-page screenshot
3. Saves it with a descriptive filename
4. Attaches it to the Cucumber report (visible in HTML reports)

Here is the part that saves you work: webship-js 2.0 ships this behavior built in. WEBSHIP_SCREENSHOT_ON_FAILED defaults to true, captures land in WEBSHIP_SCREENSHOT_DIR (./screenshots by default) with the failed_ prefix from WEBSHIP_SCREENSHOT_FAILED_PREFIX, and filenames follow WEBSHIP_SCREENSHOT_PATTERN, '{datetime}.{feature_file}.feature_{step_line}.{ext}', so the filename alone tells you which feature and which step line died. Write a custom hook like the

one above only when you need behavior the env vars cannot express; otherwise you are maintaining code the framework already gives you. Two related switches worth knowing: WEBSHIP_SCREENSHOT_ON_EVERY_STEP for full step-by-step capture (or the @screenshots tag to get the same for a single scenario), and WEBSHIP_SCREENSHOT_PURGE to clear the directory at the start of each run.

At scale: Failure screenshots only help if people can find them. Publish the screenshots directory as a CI artifact on every failed job, and keep the filename pattern identical across all your projects, when 40 suites run nightly, a triager who sees `failed_2026-06-10...login.feature_12.png` knows the feature and step before opening anything. The QA lead's first question on a red run changes from "can anyone reproduce this?" to "look at the picture: real bug or environment hiccup?" That one change is worth more than any flaky-test policy you will ever write.

Recipe 149 Full-Page Screenshots

Problem: A stakeholder asks what the new pricing page actually looks like on staging, and the only honest answer is "run the suite and watch." Worse: a failure screenshot shows a perfectly fine viewport while the broken section sits 2000 pixels below the fold. Default screenshots capture only what a user would see without scrolling, which is often exactly the part of the page that was fine.

Solution:

```
// Custom step definition for on-demand screenshots
const { When } = require('@cucumber/cucumber');
const path = require('path');

When('I take a screenshot', async function () {
  const screenshotDir = path.join(process.cwd(), 'screenshots');
  const timestamp = new Date().toISOString().replace(/[:.]/g, '-');
  await this.page.screenshot({
    path: path.join(screenshotDir, `screenshot-${timestamp}.png`),
    fullPage: true,
  });
});

When('I take a screenshot named {string}', async function (name) {
  const screenshotDir = path.join(process.cwd(), 'screenshots');
  const safeName = name.replace(/[^a-zA-Z0-9]/g, '-').toLowerCase();
  await this.page.screenshot({
    path: path.join(screenshotDir, `${safeName}.png`),
    fullPage: true,
  });
});
```

```
Feature: Page Screenshots
  Scenario: Capture dashboard for review
    Given I am a logged in user with the "admin" user
    When I go to "/admin/dashboard"
    And wait for AJAX to finish
    And I take a screenshot named "admin-dashboard"
    Then I should see "Dashboard"

  Scenario: Capture full product listing page
    Given I am on homepage
    When I go to "/products"
    And wait for AJAX to finish
    And wait for the page to load
    And I take a screenshot named "product-listing"
    Then I should see "Products"
```

Full Page vs Viewport:

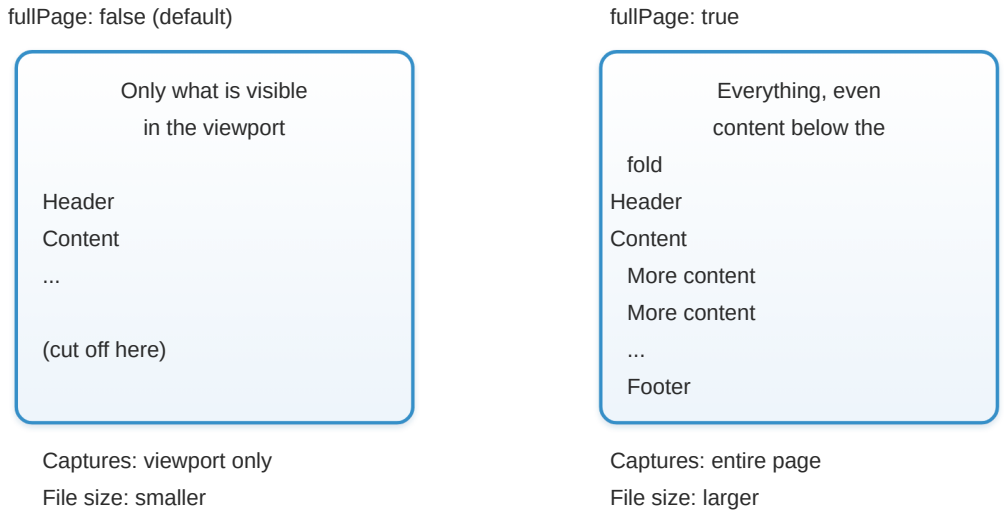


FIGURE 14.3 *Full-Page Screenshots (Recipe 149)*

How It Works:

With `fullPage: true`, Playwright scrolls the page internally, stitches the result into a single PNG, and restores the scroll position, the page under test does not stay scrolled. The built-in capture system has the same switch: set WEBSHIP_SCREENSHOT_FULLSCREEN and every automatic screenshot, failure shots included, becomes full-page. Two caveats are worth knowing before you flip it everywhere. `position: fixed` elements (sticky headers, chat widgets) repeat or stretch in stitched captures, and lazy-loaded images can appear as placeholders because the stitching scroll outruns the loader. For evidence shots of long pages both are usually acceptable; for visual regression baselines (Recipe 152) they are noise you have to plan around, which is one more argument for element-level baselines.

Recipe 150 Element-Specific Screenshots

Problem: The full-page capture of your analytics dashboard is 14,000 pixels tall, and the one chart you care about is somewhere in the middle of it. When the design team asks "send me how the revenue chart renders with real data," you want a tight crop of that component, not a scavenger hunt through a giant PNG that is mostly navigation and footer.

Solution:

```
// Custom step definition for element screenshots
const { When } = require('@cucumber/cucumber');
const path = require('path');

When('I take a screenshot of the {string} element named {string}', async
function (selector, name) {
  const screenshotDir = path.join(process.cwd(), 'screenshots');
  const safeName = name.replace(/[^a-zA-Z0-9]/g, '-').toLowerCase();
  const element = this.page.locator(selector).first();
  await element.screenshot({
    path: path.join(screenshotDir, `${safeName}.png`),
  });
});
```

```
Feature: Element Screenshots
  Scenario: Capture only the login form
    Given I am on homepage
    When I go to "/login"
    And I take a screenshot of the ".login-form" element named "login-
form"

  Scenario: Capture a specific chart
    Given I am a logged in user with the "admin" user
    When I go to "/admin/analytics"
    And wait for AJAX to finish
    And I take a screenshot of the "#revenue-chart" element named
"revenue-chart"
```

How It Works:

`element.screenshot()` waits for the element to reach a stable state (visible, done animating), scrolls it into view, then clips the capture to the element's bounding box. The result is deterministic in a way full-page shots are not: a tight crop of one component does not change when an unrelated banner above it grows by 20 pixels. That stability pays off later, element shots make far better visual-regression baselines than full pages, because the diff only covers pixels the component actually owns. Note the `.first()` in the step: if your selector matches several elements, you capture the first match silently, so prefer ids or specific data attributes for capture targets.

Recipe 151 Organizing Screenshot Archives

Problem: Three months into a project, your screenshots folder holds four thousand files with timestamps for names. A developer asks for "the screenshot from when the checkout test broke last Tuesday" and you genuinely cannot find it. Flat directories work for a week; after that, screenshots without structure are write-only data, captured at real cost, never looked at again.

Solution:

```
// Organized screenshot directory structure
const { After, Before, Status } = require('@cucumber/cucumber');
const path = require('path');
const fs = require('fs');

// Create run-specific directory before all tests
const runTimestamp = new Date().toISOString().replace(/[:.]/g, '-
').slice(0, 19);
const runDir = path.join(process.cwd(), 'screenshots', runTimestamp);

Before(function () {
  if (!fs.existsSync(runDir)) {
    fs.mkdirSync(runDir, { recursive: true });
  }
});

After(async function (scenario) {
  if (scenario.result.status === Status.FAILED) {
    const featureName = scenario.pickle.uri
      .split('/').pop()
      .replace('.feature', '');
    const scenarioName = scenario.pickle.name
      .replace(/[^a-zA-Z0-9]/g, '-')
      .toLowerCase();

    const featureDir = path.join(runDir, featureName);
    if (!fs.existsSync(featureDir)) {
      fs.mkdirSync(featureDir, { recursive: true });
    }

    await this.page.screenshot({
      path: path.join(featureDir, `${scenarioName}.png`),
      fullPage: true,
    });
  }
});
```

Resulting directory structure:

```
screenshots/
├── 2024-01-15T10-30-45/
│   ├── login/
│   │   ├── login-with-wrong-password.png
│   │   └── login-with-expired-session.png
│   ├── dashboard/
│   │   └── admin-widget-loading-error.png
│   └── forms/
│       └── file-upload-too-large.png
├── 2024-01-16T09-15-22/
│   └── login/
│       └── login-with-wrong-password.png
└── latest -> 2024-01-16T09-15-22/
```

FIGURE 14.4 *Organizing Screenshot Archives (Recipe 151)*

How It Works:

The Before hook computes one run-level directory per execution, so every scenario in a run files its captures under the same timestamp, exactly what you want when comparing “last night” against “tonight”. Inside it, failures group by feature file, mirroring how your team already thinks about the suite. Before building this by hand, check whether the built-in naming gets you there: WEBSHIP_SCREENSHOT_PATTERN supports {datetime}, {feature_file}, and {step_line} tokens, WEBSHIP_SCREENSHOT_INFO_TYPES controls which details (url, feature, step, datetime) are recorded alongside captures, and WEBSHIP_SCREENSHOT_PURGE handles cleanup at run start. The custom hook earns its keep when you need real directories, per-run, per-feature, rather than structured filenames.

At scale: Storage is the silent bill. Forty suites times nightly runs times full-page PNGs adds up to gigabytes a week, and nobody ever deletes screenshots by hand. Decide retention in the pipeline, not in test code: keep failure screenshots for 30 days as CI artifacts, purge passing-run captures

with WEBSHIP_SCREENSHOT_PURGE, and let the run-timestamp directory name double as the key your cleanup job sorts by. Your platform team will find the unbounded screenshots bucket eventually, better they find a policy.

Recipe 152 Visual Regression with Screenshot Comparison

Problem: A CSS refactor ships, every functional test passes, and the next morning the product owner asks why the homepage hero is purple. Functional assertions check that elements exist and contain the right text; none of them noticed that a shared variable changed and restyled half the site. You need a test that fails when pixels change, on purpose, against a baseline somebody approved.

Solution:

```
// tests/step-definitions/custom-visual-regression.js
const { Then } = require('@cucumber/cucumber');
const path = require('path');
const fs = require('fs');
const assert = require('assert');
const { PNG } = require('pngjs');
const pixelmatch = require('pixelmatch');

Then('the page should match the {string} baseline', async function
(baselineName) {
  const baselineDir = path.join(process.cwd(), 'screenshots',
'baselines');
  const diffDir = path.join(process.cwd(), 'screenshots', 'diffs');
  const currentDir = path.join(process.cwd(), 'screenshots', 'current');

  for (const dir of [baselineDir, diffDir, currentDir]) {
    if (!fs.existsSync(dir)) fs.mkdirSync(dir, { recursive: true });
  }

  const safeName = baselineName.replace(/[^a-zA-Z0-9]/g, '-
').toLowerCase();
  const baselinePath = path.join(baselineDir, `${safeName}.png`);
  const currentPath = path.join(currentDir, `${safeName}.png`);
```

```
  // Capture current screenshot
  await this.page.screenshot({ path: currentPath, fullPage: true });

  // If no baseline exists, create it
  if (!fs.existsSync(baselinePath)) {
    fs.copyFileSync(currentPath, baselinePath);
    return; // First run establishes baseline
  }

  // Compare images
  const baseline = PNG.sync.read(fs.readFileSync(baselinePath));
  const current = PNG.sync.read(fs.readFileSync(currentPath));
  const { width, height } = baseline;
  const diff = new PNG({ width, height });

  const mismatchedPixels = pixelmatch(
    baseline.data, current.data, diff.data,
    width, height,
    { threshold: 0.1 } // Tolerance for anti-aliasing
  );

  // Save diff image
  const diffPath = path.join(diffDir, `${safeName}-diff.png`);
  fs.writeFileSync(diffPath, PNG.sync.write(diff));

  const mismatchPercentage = (mismatchedPixels / (width * height)) * 100;

  assert(mismatchPercentage < 1,
    `Visual regression detected: ${mismatchPercentage.toFixed(2)}% pixels
differ. ` +
    `See diff: ${diffPath}`);
});
```

```
Feature: Visual Regression
  @visual
  Scenario: Homepage looks correct
    Given I am on homepage
    And wait for the page to load
    Then the page should match the "homepage" baseline

  @visual
  Scenario: Login page looks correct
    Given I am on homepage
    When I go to "/login"
    And wait for the page to load
    Then the page should match the "login-page" baseline

  @visual
  Scenario: Dashboard looks correct
    Given I am a logged in user with the "admin" user
    When I go to "/admin/dashboard"
    And wait for AJAX to finish
    And wait for the page to load
    Then the page should match the "admin-dashboard" baseline
```

How It Works:

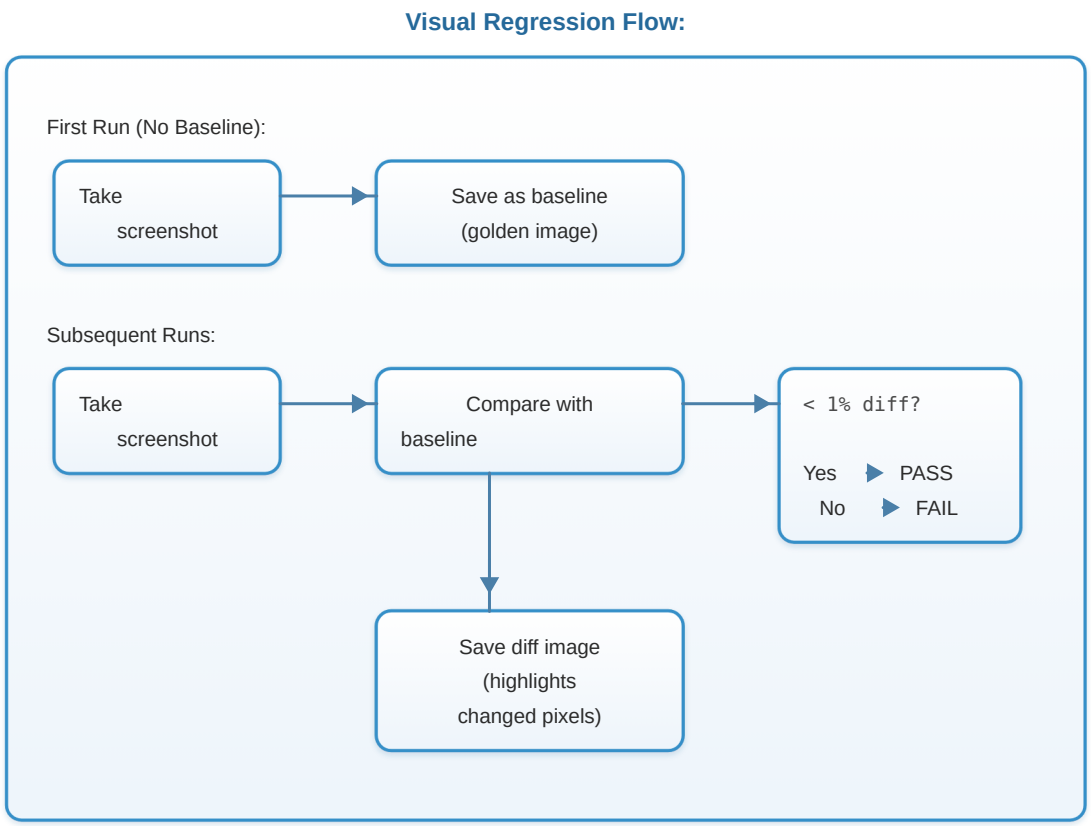


FIGURE 14.5 *Visual Regression with Screenshot Comparison (Recipe 152)*

The `pixelmatch` library compares two PNG images pixel by pixel and generates a diff image highlighting the differences in red. A threshold of 0.1 allows for minor anti-aliasing differences.

The comparison is brutally literal, which is both the power and the trap. Font hinting, anti-aliasing, and GPU rasterization differ between machines, a baseline captured on a developer's laptop will "fail" against a CI runner's capture even though nothing changed. The fix is a rule, not code: baselines are generated in the same environment that compares against them. In practice, CI generates and CI compares, on a pinned browser image; the moment your platform team bumps that image, expect to re-baseline.

webship-js's smart-wait does quiet work for you here. A screenshot taken while a spinner is mid-rotation or an AJAX call is in flight produces a different image every run, but because smartSettle waits for window.__webshipAjaxCount to hit zero and the DOM to stay quiet for 250ms after each step, your capture lands on a settled page. For regions that are genuinely dynamic (dates, ads, "5,231 users online"), either screenshot stable components instead (Recipe 150) or mask the dynamic region before comparing. And treat the 1% threshold in the assert as a starting point, not a law: tighten it for marketing pages where a stray pixel matters, loosen it for data-heavy admin screens.

At scale: Visual regression across many projects lives or dies on baseline logistics. Keep baselines per browser and per viewport (a chromium-1280 baseline will never match a webkit capture), store them in the repo or an artifact bucket keyed by branch, and make "update baselines" a deliberate, reviewed step, a CI job someone triggers after a design change, never an automatic overwrite. When a shared design system bumps a color token, every consuming project's visual suite goes red at once. That is not noise; that is the system telling you exactly which products the change reached, before any customer does.

Updating Baselines:

When your design intentionally changes, update the baselines:

```
# Delete old baselines to regenerate
rm -rf screenshots/baselines/
# Run tests -- new baselines will be created
npx cucumber-js --tags "@visual"
```

Recipe 153 Attaching Screenshots to Reports

Problem: Your screenshots live on the CI runner's disk; your test report lives in the browser of whoever opens it. A delivery manager reading the HTML report sees "Then I should see Dashboard, failed" and has to file a ticket asking what the screen showed, because the picture that answers her question sits in a job artifact she does not know exists. Evidence that is not in the report might as well not exist.

Solution:

```
// In your After hook or custom step
const { After, Status } = require('@cucumber/cucumber');

After(async function (scenario) {
  // Always attach a screenshot to the report (pass or fail)
  const screenshot = await this.page.screenshot({ fullPage: true });
  this.attach(screenshot, 'image/png');

  // Or conditionally on failure
  if (scenario.result.status === Status.FAILED) {
    const screenshot = await this.page.screenshot({ fullPage: true });
    this.attach(screenshot, 'image/png');
  }
});
```

How It Works:

`this.attach(buffer, mimeType)` is a Cucumber built-in method that attaches binary data to the current scenario's report output. When using `cucumber-html-reporter`, the screenshot appears inline in the HTML report alongside the scenario steps.

Under the hood, the attachment is base64-encoded into the message stream and lands inside tests/reports/cucumber_report.json, the same file webship-js's bin/generate-reports.js reads to build the HTML and PDF reports. That has a practical consequence: attach-on-every-step across a 400-scenario suite produces a JSON file in the hundreds of megabytes and an HTML report browsers struggle to open. Attach on failure by default; attach on success only for the handful of scenarios where a passing picture is itself the deliverable.

At scale: This is the recipe that turns a test run into evidence a non-engineer can read. Generate the report with metadata that answers a manager's first three questions, `bin/generate-reports.js` takes `--app-version`, `--env`, `--browser`, `--platform`, and arbitrary `--metadata Key=Value` flags, and the failed scenario with its inline screenshot becomes a self-contained artifact: what ran, where, against which build, and what the screen showed. When the PMO asks "is release X safe?", a PDF report (`--format pdf`) with embedded failure screenshots is an answer. A green checkbox in CI is not.

HTML Report:

```
Scenario: Login with wrong password                          ✗ FAIL

Given I am on homepage                                        ✓
When I go to "/login"                                         ✓
And I fill in "Email" with "admin@e.c"                        ✓
And I fill in "Password" with "wrong"                         ✓
And I press "Sign In"                                         ✓
Then I should see "Dashboard"                                 ✗

  [Screenshot: shows login page with error
   message "Invalid credentials"]
```

FIGURE 14.6 *Attaching Screenshots to Reports (Recipe 153)*

Screenshot Testing Quick Reference

WHAT YOU WANT	APPROACH
Screenshot on failure	After hook with Status.**FAILED**
Full-page screenshot	page.screenshot({fullPage:true})
Element screenshot	element.screenshot()
Attach to report	this.attach(buffer, 'image/png')
Visual regression	pixelmatch baseline comparison
Organize by date	Timestamped subdirectories
Organize by feature	Feature-name subdirectories
Update baselines	Delete baselines/ and rerun

Chapter Summary

- Functional assertions miss visual bugs (overlaps, invisible text, broken layouts)
- Capture screenshots automatically on test failure via After hooks
- Use `fullPage: true` to capture content below the fold
- Element screenshots capture specific components (charts, forms, tables)
- Organize screenshots by run date and feature name
- Visual regression testing compares current screenshots against saved baselines
- `pixelmatch` provides pixel-by-pixel comparison with configurable tolerance
- `this.attach()` embeds screenshots in Cucumber HTML reports
- First run creates baselines; subsequent runs compare against them
- Update baselines when design intentionally changes

Next: Accessibility Testing, ensuring your application is usable by everyone.

Chapter 15: Accessibility Testing

Accessibility is not a feature. It is a fundamental quality of software. When your login form cannot be navigated with a keyboard, you have excluded every user who relies on one. When your images lack alt text, screen readers announce nothing. When your color contrast is insufficient, users with low vision cannot read your content.

Automated accessibility testing catches the most common issues before they reach production. This chapter shows you how to integrate accessibility checks into your webship-js test suite.

Understanding Web Accessibility

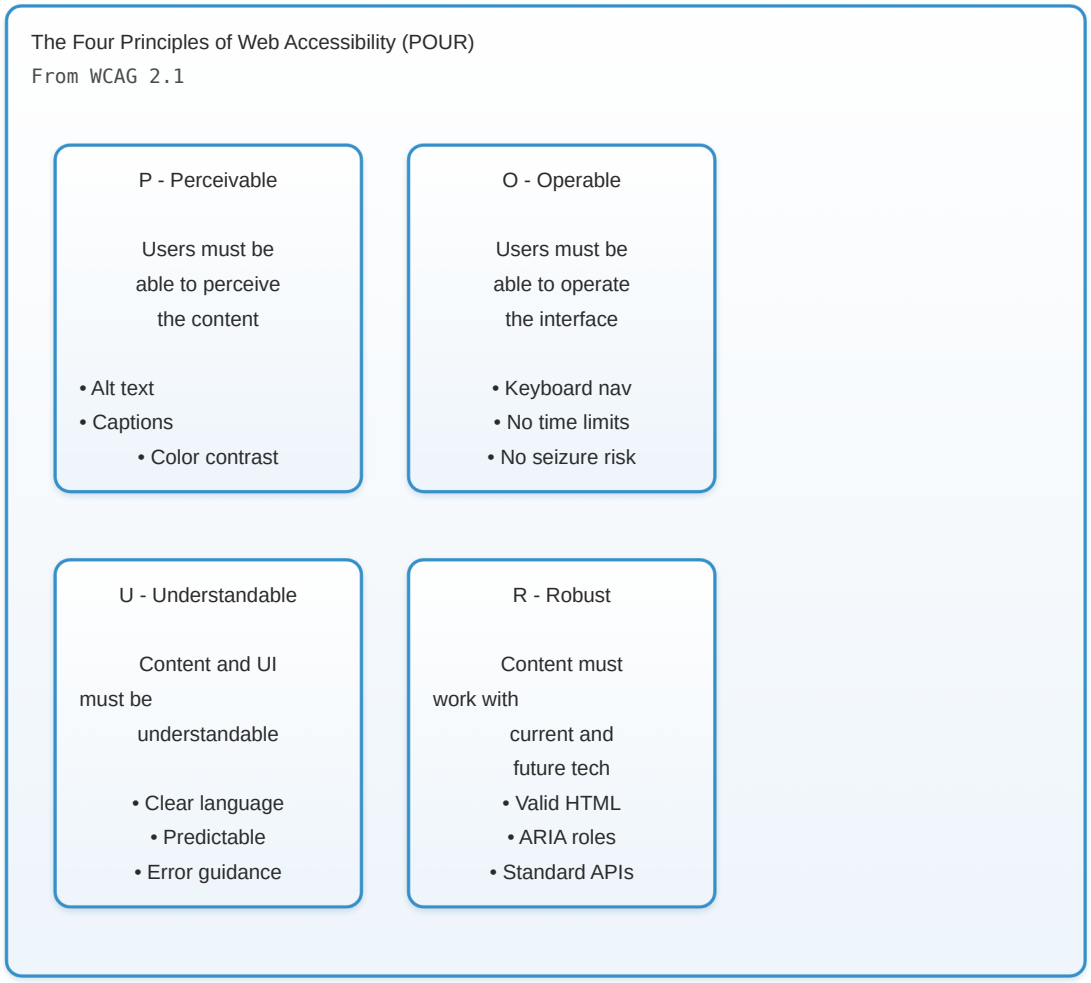


FIGURE 15.1 *Understanding Web Accessibility*

Recipe 154 Testing Keyboard Navigation

Problem: A user who cannot use a mouse Tabs into your login form and gets stuck, focus jumps from Email straight into a hidden newsletter field, then vanishes into an off-screen carousel. No functional test notices, because functional tests click. The first time anyone discovers it is an accessibility audit, a procurement questionnaire, or a complaint, and every one of those costs more than the test would have.

Solution:

```
// tests/step-definitions/custom-keyboard.js
const { When, Then } = require('@cucumber/cucumber');
const assert = require('assert');

When('I press the Tab key', async function () {
  await this.page.keyboard.press('Tab');
});

When('I press the Tab key {int} times', async function (count) {
  for (let i = 0; i < count; i++) {
    await this.page.keyboard.press('Tab');
  }
});

When('I press the Enter key', async function () {
  await this.page.keyboard.press('Enter');
});

When('I press the Escape key', async function () {
  await this.page.keyboard.press('Escape');
});

Then('the focused element should be {string}', async function
(expectedText) {
  const focused = await this.page.evaluate(() => {
    const el = document.activeElement;
    return el ? (el.textContent || el.value || el.placeholder ||
el.getAttribute('aria-label') || '').trim() : '';
  });
  assert(focused.includes(expectedText),
```

```
    `Expected focused element to contain "${expectedText}", but got
"${focused}"`);
});

Then('the focused element should have role {string}', async function
(expectedRole) {
  const role = await this.page.evaluate(() => {
    const el = document.activeElement;
    return el ? (el.getAttribute('role') || el.tagName.toLowerCase()) :
'';
  });
  assert.strictEqual(role, expectedRole,
    `Expected focused element role "${expectedRole}", but got "${role}"`);
});
```

```
Feature: Keyboard Navigation
  As a keyboard-only user
  I want to navigate the site without a mouse
  So that I can use all features

  Scenario: Tab through the login form
    Given I am on homepage
    When I go to "/login"
    And I press the Tab key
    Then the focused element should be "Email"
    When I press the Tab key
    Then the focused element should be "Password"
    When I press the Tab key
    Then the focused element should be "Sign In"

  Scenario: Submit form with Enter key
    Given I am on homepage
    When I go to "/login"
    And I fill in "Email" with "admin@example.com"
    And I fill in "Password" with "AdminPass123"
    And I press the Enter key
    And wait for AJAX to finish
    Then I should see "Dashboard"

  Scenario: Close modal with Escape key
```

```
    Given I am a logged in user with the "admin" user
    When I go to "/admin/users"
    And I press "Add User"
    And wait for the modal to appear
    Then I should see a modal dialog
    When I press the Escape key
    And wait for the modal to disappear
    Then I should not see a modal dialog

  Scenario: Navigate menu with keyboard
    Given I am on homepage
    When I press the Tab key 3 times
    And I press the Enter key
    And wait for AJAX to finish
    Then I should see "Products"
```

How It Works:

`page.keyboard.press('Tab')` dispatches a real, trusted key event through the browser's input pipeline, the browser itself decides where focus lands, following DOM order, tabindex, and visibility exactly as it would for a real user. That is the point: you are testing the browser's actual focus order, not your assumption of it. The assertion step then reads `document.activeElement` inside the page and checks its text, value, placeholder, or aria-label, the same accessible-name sources a screen reader consults. The fallback chain in that step is deliberate: a focused element that has none of those is itself an accessibility bug, and the empty-string failure message will tell you so.

The Enter-key scenario covers a behavior teams break constantly: implicit form submission. Wire the click handler to the button instead of the submit event to the form, and Enter-to-submit dies, only a keyboard test catches it. And because webship-js's AfterStep smartSettle runs after each key press, focus assertions land after any focus-shifting JavaScript has finished, not mid-transition.

At scale: Keyboard steps like these are exactly the custom code that gets copy-pasted into every project and then drifts. Keep them in one shared step-definition package, tag the scenarios @a11y, and run that tag everywhere. When a procurement questionnaire asks "do you test keyboard accessibility?", pointing at the same green tag across 20 products is a better answer than any policy document your PMO can draft.

Recipe 155 Testing Focus Management

Problem: A modal opens and focus stays on the button behind it, so a keyboard user's next Tab walks the page underneath a dialog they cannot see past. Or a form fails validation and the error message renders 800 pixels above the focused submit button, announced to no one. Focus management bugs are invisible in a mouse-driven sprint demo and crippling for everyone who navigates by keyboard.

Solution:

```
Feature: Focus Management
  Scenario: Focus moves to modal when opened
    Given I am a logged in user with the "admin" user
    When I go to "/admin/users"
    And I press "Add User"
    And wait for the modal to appear
    # Focus should be on the first input in the modal
    Then the focused element should be "First Name"

  Scenario: Focus moves to error on validation failure
    Given I am on homepage
    When I go to "/login"
    And I press "Sign In"
    # Focus should jump to the first error
    Then I should see "Email is required"

  Scenario: Focus returns after modal closes
    Given I am a logged in user with the "admin" user
    When I go to "/admin/users"
    And I press "Add User"
    And wait for the modal to appear
    When I close the modal
    And wait for the modal to disappear
    # Focus should return to the button that opened the modal
    Then the focused element should be "Add User"
```

How It Works:

These scenarios reuse the focused-element step from Recipe 154 to pin down the three moments WCAG cares about most: focus moves into a dialog when it opens, focus lands somewhere useful when an error appears, and focus returns to the trigger when the dialog closes. The third is the one most commonly broken, close a modal and focus silently resets to `<body>`, stranding a keyboard user back at the top of the page with no idea where they were. webship-js's auto-settle earns its keep here: the AfterStep smartSettle waits for the modal's open and close animations (visible to it through window.__webshipPendingTimers and the DOM MutationObserver) before the next step runs, so you assert focus on the final state, never mid-transition.

Recipe 156 Testing ARIA Labels and Roles

Problem: The page looks perfect and reads as garbage. A screen reader announces your icon-only search button as "button", your hero image as its filename, and offers no landmarks to jump between, so navigating means listening to every element, in order, every visit. None of this shows up visually, which is exactly why it survives review after review until someone who depends on it files the bug.

Solution:

```
// tests/step-definitions/custom-aria.js
const { Then } = require('@cucumber/cucumber');
const assert = require('assert');

Then('the {string} element should have aria-label {string}', async
function (selector, label) {
  const ariaLabel = await
this.page.locator(selector).first().getAttribute('aria-label');
  assert.strictEqual(ariaLabel, label,
    `Expected aria-label "${label}", got "${ariaLabel}"`);
});

Then('the page should have a {string} landmark', async function (role) {
  const count = await this.page.locator(`[role="${role}"],
${role}`).count();
  assert(count > 0, `No element with role="${role}" found on the page`);
});

Then('all images should have alt text', async function () {
  const images = await this.page.locator('img').all();
  for (const img of images) {
    const alt = await img.getAttribute('alt');
    const src = await img.getAttribute('src');
    assert(alt !== null && alt !== undefined,
      `Image "${src}" is missing alt attribute`);
  }
});
```

```
Feature: ARIA Compliance
  Scenario: Page has proper landmarks
    Given I am on homepage
    Then the page should have a "navigation" landmark
    And the page should have a "main" landmark

  Scenario: All images have alt text
    Given I am on homepage
    When I go to "/products"
    And wait for AJAX to finish
    And wait for the page to load
    Then all images should have alt text

  Scenario: Interactive elements are labeled
    Given I am on homepage
    When I go to "/login"
    Then the "#search-icon" element should have aria-label "Search"
```

How It Works:

The landmark step accepts both spellings of the same semantics, explicit `role="navigation"` attributes and native elements like `<nav>` that carry the role implicitly, which is why the locator queries `[role="${role}"], ${role}`. The alt-text step is stricter than it looks: it asserts the attribute *exists*, not that it contains text, because `alt=""` is the correct way to mark a decorative image, while a missing attribute makes screen readers fall back to announcing the filename. Before writing more checks like these, look at what ships in the box: webship-js 2.0 includes structural accessibility steps for landmarks, heading hierarchy, form labels, skip links, tabindex order, and ARIA references. Custom steps should encode your project's specific conventions, not re-implement the basics.

At scale: ARIA conventions are a team agreement, and agreements need enforcement to survive the third project you onboard. Put the landmark and labeling scenarios in a shared feature file every new project starts from, they are application-agnostic by design, and they become the cheapest accessibility regression net you will ever run: no test data, no login, just structure. New project, day one, already covered.

Recipe 157 Testing Color Contrast

Problem: A brand refresh lightens the gray used for body text, and suddenly anyone with low vision, or anyone reading a phone in sunlight, cannot make out your content. WCAG AA requires 4.5:1 contrast for normal text and 3:1 for large text, and no human reviewer can eyeball a ratio. Contrast is the most common accessibility failure on the web precisely because it never looks "broken" to the people who built it.

Solution:

webship-js has a built-in CSS assertion step that can verify computed styles:

```
Feature: Color Contrast
  Scenario: Text is readable against background
    Given I am on homepage
    # Verify element has specific CSS properties
    Then the element "#main-heading" should have CSS "color" of "rgb(33,
37, 41)"
    And the element "#main-heading" should have CSS "font-size" of "32px"
```

For automated contrast ratio checking you do not need to build anything: webship-js 2.0 bundles axe-core and ships the audit as built-in steps, `Then the page should pass an accessibility audit at level "AA"`, `Then the page should have no critical accessibility violations`, and `Then print accessibility violations` for dumping the findings into the test output. Use those first. The custom version below is still worth reading, because it shows what the built-in steps do under the hood and how you would tune the failure rules if your project needs different thresholds:

```
// tests/step-definitions/custom-a11y.js
const { Then } = require('@cucumber/cucumber');
const assert = require('assert');

Then('the page should pass accessibility checks', async function () {
  // Inject axe-core
  await this.page.addScriptTag({
    url: 'https://cdnjs.cloudflare.com/ajax/libs/axe-
```

```
core/4.7.0/axe.min.js'
  });

  // Run axe
  const results = await this.page.evaluate(async () => {
    return await axe.run();
  });

  const violations = results.violations;
  if (violations.length > 0) {
    const report = violations.map(v =>
      `${v.impact}: ${v.description} (${v.nodes.length} instances)`
    ).join('\n');

    assert.fail(`Accessibility violations found:\n${report}`);
  }
});

Then('the page should have no critical accessibility issues', async
function () {
  await this.page.addScriptTag({
    url: 'https://cdnjs.cloudflare.com/ajax/libs/axe-
core/4.7.0/axe.min.js'
  });

  const results = await this.page.evaluate(async () => {
    return await axe.run();
  });

  const critical = results.violations.filter(
    v => v.impact === 'critical' || v.impact === 'serious'
  );

  if (critical.length > 0) {
    const report = critical.map(v =>
      `[${v.impact}] ${v.description}\n  ${v.helpUrl}`
    ).join('\n\n');

    assert.fail(`Critical accessibility violations:\n${report}`);
  }
});
```

```
Feature: Accessibility Audit
  @a11y
  Scenario: Homepage passes accessibility checks
    Given I am on homepage
    And wait for the page to load
    Then the page should pass an accessibility audit at level "AA"

  @a11y
  Scenario: Login page has no critical issues
    Given I am on homepage
    When I go to "/login"
    And wait for the page to load
    Then the page should have no critical accessibility violations

  @a11y
  Scenario: Dashboard passes accessibility checks
    Given I am a logged in user with the "admin" user
    When I go to "/admin/dashboard"
    And wait for AJAX to finish
    And wait for the page to load
    Then the page should have no critical accessibility violations
```

How It Works:

axe-core is the industry standard accessibility testing engine. It checks for:

CATEGORY	WHAT IT CHECKS
Color contrast	Text vs background ratio (WCAG AA/AAA)
Alt text	Images, icons, inputs have labels
ARIA roles	Valid roles, required properties
Form labels	Inputs associated with labels
Headings	Correct heading hierarchy (h1→h2→h3)
Landmarks	Page has navigation, main, etc.
Links	Link text is descriptive (not "click here")
Tables	Tables have headers and captions
Focus	Focus indicators visible
Language	Page has lang attribute

webship-js's built-in audit steps run this engine against the live, rendered page, which matters for contrast specifically, because the real ratio depends on computed styles, inherited backgrounds, and overlays that only exist at runtime. The level parameter ("A", "AA", "AAA") selects which WCAG rule set applies; the no-critical-violations step filters by impact instead, which lets a legacy page run a looser gate while new work holds the full line. And because the engine ships bundled with the framework, the audit works on air-gapped CI runners where the CDN injection in the custom version above would fail outright, one more reason to prefer the built-ins. When a run goes red, `Then print accessibility violations` writes each finding with its impact and help URL into the test output, which lands in the JSON report and therefore in the HTML report your team actually reads.

At scale: One audit step, the same level “AA” string, in every project, that consistency is what turns accessibility from a per-project opinion into a portfolio-wide number. Tag the audits @a11y, run them in every pipeline, and stamp the result into report metadata (`--metadata A11y=AA`) so the cross-project dashboard can show it. Fair warning: the first nightly run across 20 legacy sites will be ugly. Gate on critical violations first, ratchet each project up to the full AA audit as it cleans up, and the dashboard shows your PMO an improving line instead of a wall of red nobody acts on.

Recipe 158 Testing Screen Reader Content

Problem: Your form saves and a green toast says so, visually. A screen reader user hears nothing, waits, and submits again, creating a duplicate record. The cues sighted users get for free (a toast, a highlight, a spinner) have to be deliberately wired into assistive tech with aria-live regions and skip links, and deliberate wiring is exactly what disappears silently in refactors.

Solution:

```
// tests/step-definitions/custom-sr.js
const { Then } = require('@cucumber/cucumber');
const assert = require('assert');

Then('there should be a skip link', async function () {
  const skipLink = this.page.locator('a[href="#main-content"], a.skip-
link, a.skip-to-content');
  const count = await skipLink.count();
  assert(count > 0, 'No skip-to-content link found');
});

Then('the {string} element should have aria-live {string}', async function
(selector, value) {
  const ariaLive = await
this.page.locator(selector).first().getAttribute('aria-live');
  assert.strictEqual(ariaLive, value,
    `Expected aria-live="${value}", got "${ariaLive}"`);
});
```

```
Feature: Screen Reader Accessibility
  Scenario: Skip link is available
    Given I am on homepage
    Then there should be a skip link

  Scenario: Status messages are announced
    Given I am a logged in user with the "admin" user
    When I go to "/settings"
    Then the ".status-messages" element should have aria-live "polite"
    When I press "Save"
    And wait for AJAX to finish
    Then I should see "Settings saved"
```

How It Works:

The skip-link step queries the three common conventions in one locator and passes if any exists, though webship-js also ships a structural skip-link check built in, so reach for the custom version only when your projects use a selector the built-in does not know. The aria-live assertion is the one to internalize: a region marked `aria-live="polite"` makes assistive tech announce any text injected into it, which is the only way your “Settings saved” toast reaches a screen reader at all. Note the ordering in the scenario, the attribute is checked *before* triggering the save, on purpose. The live region must exist in the DOM before the message arrives, because screen readers do not announce content placed in regions created after the fact. That ordering bug is the classic way live regions fail while looking perfectly correct in the DOM inspector.

Accessibility Testing Checklist

```
Accessibility Testing Checklist for webship-js

Automated (axe-core):
  ☐ Color contrast ratios meet WCAG AA
  ☐ All images have alt text
  ☐ Form inputs have associated labels
  ☐ Page has proper landmark roles
  ☐ Heading hierarchy is correct (no skipped levels)
  ☐ Links have descriptive text
  ☐ ARIA attributes are valid
  ☐ Language attribute is set on <html>

Semi-Automated (custom steps):
  ☐ Tab order follows visual order
  ☐ All interactive elements are keyboard accessible
  ☐ Focus moves correctly after actions (modals, errors)
  ☐ Skip link is present and functional
  ☐ Modal focus trapping works
  ☐ Dynamic content updates announce to screen readers

Manual Testing (complement automated):
  ☐ Screen reader can navigate the full application
  ☐ Zoom to 200% does not break layout
  ☐ No information is conveyed by color alone
  ☐ Animation can be paused or disabled
  ☐ Touch targets are large enough on mobile
```

FIGURE 15.2 *Accessibility Testing Checklist*

Chapter Summary

- Accessibility is not optional, it is a core quality attribute
- WCAG 2.1 defines four principles: Perceivable, Operable, Understandable, Robust
- Keyboard navigation testing ensures Tab, Enter, Escape work correctly
- Focus management testing verifies focus moves after modals, errors, navigation
- ARIA labels and roles must be present for screen reader users
- axe-core provides automated accessibility auditing with severity levels
- Run accessibility checks on every page (homepage, login, dashboard, forms)
- Use `@a11y` tags to organize accessibility tests
- Color contrast must meet WCAG AA minimums (4.5:1 normal, 3:1 large)
- Automated testing catches ~30-50% of accessibility issues; manual testing covers the rest
- Skip links, aria-live regions, and proper landmarks are testable with custom steps

This concludes Part III: API and Advanced Testing.

Next: The PlaywrightWorld Deep Dive, understanding the architecture that powers webship-js.

Part IV: Architecture and Organization

Chapter 16: The PlaywrightWorld Deep Dive

Step definitions and code examples by Tasneem Natshah, from the webship-js source.

Every Cucumber scenario starts with a fresh World object. This World carries the state that steps share during a single scenario, the browser, the page, the URL, the wait times, and any custom data you need. In webship-js, that World is `PlaywrightWorld`, and understanding it is the key to writing powerful custom steps.

This chapter takes you inside PlaywrightWorld: what it does, how it works, and how to extend it for your project's specific needs.

What Is the World Object?

In Cucumber, the World is the `this` context inside every step definition. When you write:

```
Given('I am on homepage', async function () {
  await this.page.goto(this.launchUrl);
});
```

The `this` is the World object. Every step in a single scenario shares the same World instance, which means they share the same `this.page`, `this.launchUrl`, and any other properties.

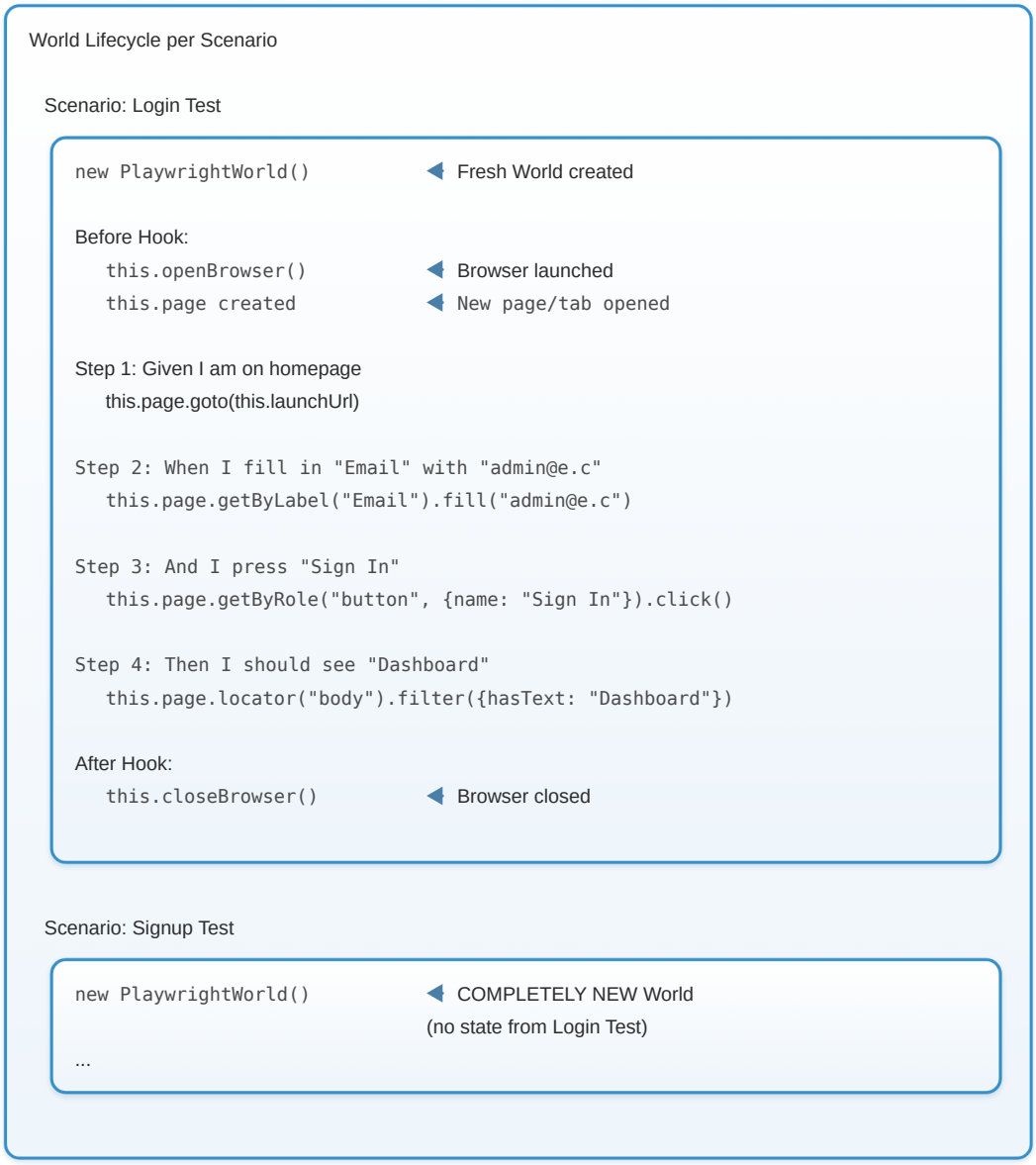


FIGURE 16.1 *What Is the World Object?*

Critical concept: Each scenario gets a **new** World instance. No state carries over between scenarios. This is how Cucumber ensures test isolation.

It is also a useful debugging lens. When a scenario passes alone but fails in the full suite, the leak is almost never in the World, it *cannot* be, because the World is rebuilt every time. The state is leaking through something outside it: the application database, a shared user account, a file on disk. Knowing what the World guarantees tells you where not to waste your afternoon looking.

PlaywrightWorld Internals

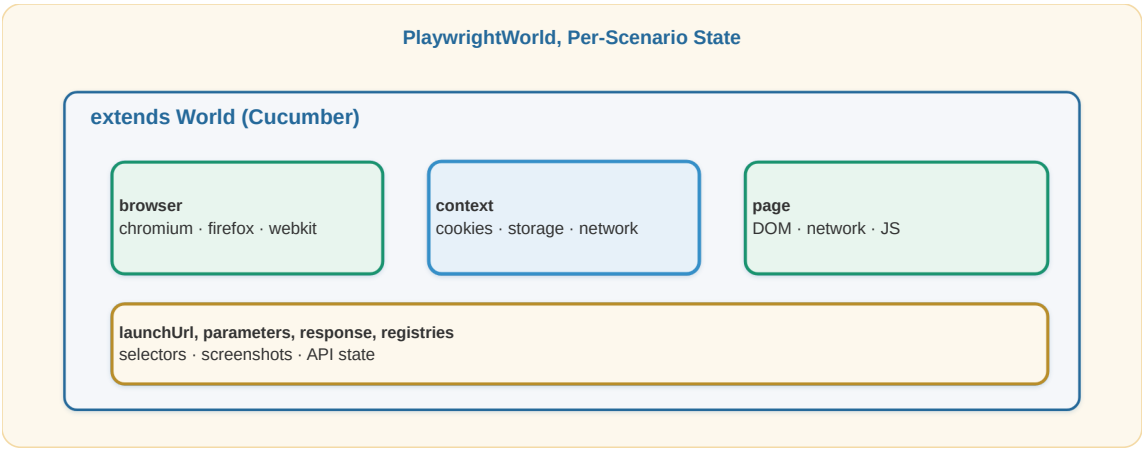


FIGURE 16.2 *PlaywrightWorld Internals*

Here is the actual PlaywrightWorld class from webship-js:

```
class PlaywrightWorld extends World {
  constructor(options) {
    super(options);
    this.launchUrl = this.parameters.launchUrl;
    this.minWaitTime = this.parameters.minWaitTime;
    this.playwrightBrowser = null;
    this.context = null;
    this.page = null;
    this.assetsFolder = path.join(__dirname, '../assets/');
  }

  async openBrowser() {
    const { browser: browserName, launchOptions, contextOptions } =
playwrightConfig;
    this.playwrightBrowser = await
playwright[browserName].launch(launchOptions);
    this.context = await
this.playwrightBrowser.newContext(contextOptions);
    this.page = await this.context.newPage();
  }

  async closeBrowser() {
    if (this.playwrightBrowser) {
      await this.playwrightBrowser.close();
      this.playwrightBrowser = null;
      this.context = null;
      this.page = null;
    }
  }
}
```

Where do the constructor values come from? `this.parameters` is the worldParameters block from cucumber.js. In webship-js 2.0 the stock profile sets launchUrl from the LAUNCH_URL environment variable (default http://localhost:8080), minWaitTime.page to 3000ms, and a selectors registry: named CSS and XPath selectors auto-loaded from JSON files in tests/selectors/, plus the breakpoints table and the 60px scroll offset the selector steps use. openBrowser() reads playwright.config.ts, which is itself environment-driven, BROWSER picks chromium, firefox, or

webkit; HEADLESS='false' switches to headed; SLOW_MO defaults to 0 in CI, 300 headless, 800 headed; viewport is null and ignoreHTTPSErrors is true. Nothing about the World is hard-coded, which is exactly what lets one suite run unmodified against five environments.

The context created in openBrowser() is also where webship-js installs its smart-wait instrumentation. An addInitScript hook injects three window globals into every page before any application code runs: window.__webshipAjaxCount (a fetch/XHR wrapper counting in-flight requests), window.__webshipPendingTimers (a setTimeout wrapper counting pending timers), and window.__webshipLastMutation (a MutationObserver timestamp). Every built-in step's auto-settle reads those globals through this.page. Remember that when you reach Recipe 160: replace the World and skip the stock browser lifecycle, and the entire smart-wait system silently disappears with it.

Properties breakdown:

PROPERTY	SOURCE	PURPOSE
this.launchUrl	cucumber.js	Base URL for navigation
	worldParameters	e.g., http://localhost:80
this.minWaitTime	cucumber.js	Timing delays (object):
	worldParameters	.before_scenario
		.after_scenario
		.before_step
		.after_step
		.page (page load wait)
this.playwright	openBrowser()	Playwright browser
Browser		instance (chromium/ff/wk)
this.context	openBrowser()	Browser context (like an
		incognito window)
this.page	openBrowser()	The page/tab you interact
		with in step definitions
this.assetsFolder	constructor	Path to test asset files
		(upload files, etc.)

The Playwright hierarchy:

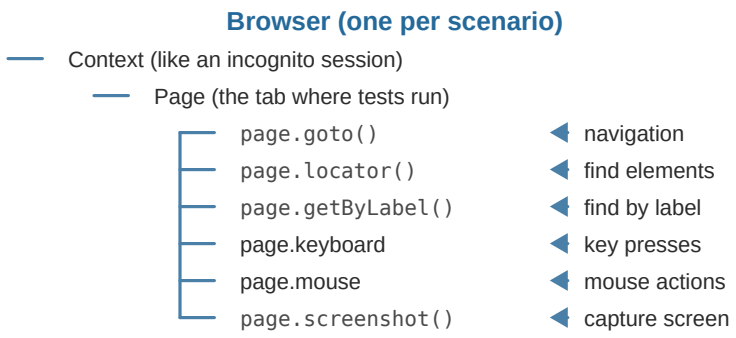


FIGURE 16.3 *PlaywrightWorld Internals*

Helper Functions

webship-js defines several helper functions that you can reuse in your custom step definitions:

buildSelector()

```
function buildSelector(attrValue, attr) {
  // "#id" or ".class"       ▶ return as-is
  // "name" (no spaces)      ▶ try tag, #id, .class, [name=], [value=], [placeholder=], [title=]
  // "two words"     ▶ try [value=], [placeholder=]
  // with attr arg     ▶ [attr="value" i] (case insensitive)
}
```

FIGURE 16.4 *buildSelector()*

INPUT	OUTPUT CSS SELECTOR
buildSelector("#myId")	#myId
buildSelector(".myClass")	.myClass
buildSelector("email")	email,#email,.email,[name=email],
	[value="email"],
	[placeholder="email"],
	[title="email"]
buildSelector("my text")	[value="my text"],
	[placeholder="my text"]
buildSelector("x","data-id")	[data-id="x" i]

fillField()

```
async function fillField(page, field, value) {
  // 1. Try: page.getByLabel(field, { exact: true })
  // 2. Try: page.getByPlaceholder(field, { exact: true })
  // 3. Fallback: page.locator(`[name="${field}"]`)
}
```

This is the label-first resolution behind the built-in `When I fill in "Label" with "value"` step. It is also a quiet design statement: tests written against visible labels survive markup refactors that would break CSS selectors, and they fail loudly when a label disappears, which is a real usability bug, not a test maintenance chore.

gotoUrl()

```
async function gotoUrl(page, url) {
  // Navigates to URL with waitUntil: 'domcontentloaded'
  // Silently handles NS_ERROR_NET_EMPTY_RESPONSE (Firefox quirk)
}
```

Recipe 159 Adding Custom Properties to the World

Problem: Your scenario creates a product in step two and needs to assert on that exact product in step six. Hardcoding the name in both steps works, until the third time a new hire copies the feature file, changes one of the two hardcoded strings, and your team spends an afternoon debugging a “flaky” test that was never flaky. Steps need a way to hand data to later steps without anyone keeping two strings in sync by hand.

Solution:

```
// tests/step-definitions/custom-world-properties.js
const { Given, When, Then } = require('@cucumber/cucumber');

Given('I create a test product named {string}', async function (name) {
  // Store the product name on the World for later steps
  this.testProductName = name;

  await this.page.goto(this.launchUrl + '/admin/products/new');
  await this.page.getByLabel('Name').fill(name);
  await this.page.getByRole('button', { name: 'Save' }).click();
  await this.page.waitForLoadState('networkidle');
});

Then('I should see the test product in the list', async function () {
  // Access the stored name from a previous step
  const locator = this.page.locator('body').filter({ hasText:
this.testProductName });
  await locator.waitFor({ timeout: 5000 });
});
```

```
Feature: Product Management
  Scenario: Create and verify product
    Given I am a logged in user with the "admin" user
    And I create a test product named "Test Widget"
    When I go to "/admin/products"
    And wait for AJAX to finish
    Then I should see the test product in the list
```

How It Works:

Since all steps in a scenario share the same World (`this`), a property set in one step is readable in every later step, no globals, no module-level variables, no files on disk. Cleanup is automatic and absolute: the next scenario constructs a brand-new PlaywrightWorld, so `this.testProductName` cannot leak forward even if the scenario fails halfway through. That guarantee is the reason to prefer World properties over the tempting alternative, a module-level `let lastProduct` in a step file survives across scenarios and quietly turns scenario *order* into hidden test logic; `this` never does.

One naming habit pays off: prefix your custom properties (`this.testProductName`, not `this.name`) so they cannot collide with what the framework already owns. page, context, launchUrl, and minWaitTime are taken, and overwriting this.page mid-scenario produces spectacular confusion.

At scale: Agree on the property names once and write them down, this.testData for created records, this.lastApiResponse for API payloads, in the shared step library every project consumes. When the same QA engineer rotates across six projects in a quarter, finding the same state in the same place each time is the difference between contributing on day one and reverse-engineering each suite's private conventions.

Recipe 160 Extending PlaywrightWorld

Problem: The same fifteen lines of login automation appear in four step files, each slightly diverged, one waits for networkidle, one does not, one still uses last quarter's password. When the login flow changes, you fix three of the four, and the suite goes intermittently red in a way nobody can explain for a week. Logic that several steps share belongs on the World as a method: written once, called everywhere.

Solution:

```
// tests/support/extended-world.js
const { setWorldConstructor, World } = require('@cucumber/cucumber');
const playwright = require('playwright');
const playwrightConfig = require(require('path').join(process.cwd(),
'playwright.config'));
const path = require('path');

class ExtendedWorld extends World {
  constructor(options) {
    super(options);
    this.launchUrl = this.parameters.launchUrl;
    this.minWaitTime = this.parameters.minWaitTime;
    this.playwrightBrowser = null;
    this.context = null;
    this.page = null;
    this.assetsFolder = path.join(__dirname, '../assets/');
  }

  async openBrowser() {
    const { browser: browserName, launchOptions, contextOptions } =
playwrightConfig;
    this.playwrightBrowser = await
playwright[browserName].launch(launchOptions);
    this.context = await
this.playwrightBrowser.newContext(contextOptions);
    this.page = await this.context.newPage();
  }

  async closeBrowser() {
    if (this.playwrightBrowser) {
```

```
      await this.playwrightBrowser.close();
      this.playwrightBrowser = null;
    }
  }

  // Custom method: login programmatically
  async loginAs(role) {
    const users = {
      admin: { email: 'admin@example.com', password: 'AdminPass123' },
      editor: { email: 'editor@example.com', password: 'EditorPass123' },
    };
    const user = users[role];
    await this.page.goto(this.launchUrl + '/login');
    await this.page.getByLabel('Email').fill(user.email);
    await this.page.getByLabel('Password').fill(user.password);
    await this.page.getByRole('button', { name: 'Sign In' }).click();
    await this.page.waitForLoadState('networkidle');
  }

  // Custom method: wait for and dismiss any toast notification
  async dismissToast() {
    const toast = this.page.locator('.toast, .snackbar,
.notification').first();
    if (await toast.isVisible()) {
      const closeBtn = toast.locator('.close, [aria-
label="Close"]').first();
      if (await closeBtn.isVisible()) await closeBtn.click();
    }
  }

  // Custom method: take a named screenshot
  async screenshot(name) {
    const dir = path.join(process.cwd(), 'screenshots');
    await this.page.screenshot({
      path: path.join(dir, `${name}.png`),
      fullPage: true,
    });
  }
}

setWorldConstructor(ExtendedWorld);
```

Then use the methods in step definitions:

```
const { Given, When } = require('@cucumber/cucumber');

Given('I am logged in as {string}', async function (role) {
  await this.loginAs(role);
});

When('I take a screenshot named {string}', async function (name) {
  await this.screenshot(name);
});
```

How It Works:

`setWorldConstructor(ExtendedWorld)` tells Cucumber to instantiate your class for every scenario, and there can be only one World constructor per suite, so this *replaces* PlaywrightWorld rather than wrapping it. That is why the example re-implements openBrowser() and closeBrowser() with the same shape: webship-js's Before and After hooks call those methods by name, and all 411 built-in steps expect this.page, this.context, and this.launchUrl to exist exactly as the stock World provides them. Keep the constructor and browser lifecycle identical and layer your methods on top; change the lifecycle and you will discover that the smart-wait instrumentation, the screenshot hooks, and every built-in step assumed the contract you just broke, and none of them fail with a message that says so.

The methods themselves follow a rule worth adopting: World methods for *operations* (loginAs, dismissToast), step definitions for *language*. Steps stay one line each, the operation is reusable from any step file, and a Gherkin reader never sees the plumbing.

For login specifically, check the built-in option before writing loginAs() everywhere: webship-js ships storage-state steps (auth.steps.js) that save a logged-in session once and restore it per scenario, the saved storageState carries cookies and localStorage, so restoring it skips the login UI entirely. loginAs() earns its place when you genuinely need to exercise the login form, or when roles switch mid-feature.

At scale: The ExtendedWorld is the single highest-leverage file to share across projects. Publish it, with your shared steps, as an internal npm package, @yourorg/test-world, and each project's tests/support/ becomes a two-line import. When the design system renames every toast selector,

you fix dismissToast() once, version-bump, and twenty suites pick it up through a dependency update on the next pipeline run, instead of twenty near-identical merge requests reviewed by twenty different people, three of which quietly diverge.

Recipe 161 Multiple World Configurations

Problem: The suite passes against localhost and fails against staging, and the “fix” on someone’s laptop was editing launchUrl in cucumber.js, which then got committed and broke everyone else’s local runs. The Tuesday before release, the suite must point at staging; Wednesday morning, at production smoke checks. Nobody should be editing config files to make that happen.

Solution:

```
// cucumber.js
const common = {
  require: ['tests/step-definitions/**/*.js'],
  format: ['@cucumber/pretty-formatter'],
};

module.exports = {
  default: {
    ...common,
    worldParameters: {
      launchUrl: 'http://localhost:8080',
      minWaitTime: {
        before_scenario: 0, after_scenario: 0,
        before_step: 0, after_step: 0, page: 3000,
      },
    },
  },

  staging: {
    ...common,
    worldParameters: {
      launchUrl: 'https://staging.example.com',
      minWaitTime: {
        before_scenario: 1000, after_scenario: 500,
        before_step: 200, after_step: 200, page: 5000,
      },
    },
  },

  production: {
    ...common,
    worldParameters: {
      launchUrl: 'https://www.example.com',
      minWaitTime: {
        before_scenario: 1000, after_scenario: 500,
        before_step: 300, after_step: 300, page: 8000,
      },
    },
  },
};
```

Run with a specific profile:

```
# Local (default)
npx cucumber-js

# Staging
npx cucumber-js --profile staging

# Production
npx cucumber-js --profile production
```

How It Works:

A Cucumber profile is just a named bundle of options. `--profile staging` swaps the entire worldParameters block without touching code, and every scenario's World constructor receives the staging values through this.parameters. The slower minWaitTime values on the remote profiles reflect a real difference, staging usually runs on shared infrastructure behind a CDN, and a page that settles in one second locally can take five. But notice what does *not* change: wait budgets are configuration, never feature-file content, so the same Gherkin runs against all three environments unmodified. That is the whole game.

Before reaching for profiles, know that webship-js's stock cucumber.js already reads LAUNCH_URL from the environment (default http://localhost:8080), `LAUNCH_URL=https://staging.example.com npx cucumber-js` covers the most common case with zero config files. Profiles earn their place when environments differ in more than the URL: wait budgets, selector files, report settings.

At scale: In CI, prefer the env var over the profile. Pipelines already manage per-environment variables and secrets, so injecting LAUNCH_URL from the pipeline environment means one pipeline template serves all 20 projects, and adding a new environment is a CI variable, not a pull request to every repo. Keep profiles for the human-shaped differences (a slow-staging wait budget a developer wants locally), let the pipeline own the URLs, and never let production URLs or credentials live in a committed config file.

Chapter Summary

- The World object is `this` in every step definition, shared within a scenario, fresh between scenarios
- PlaywrightWorld extends Cucumber's World with browser, context, page, and configuration
- Browser lifecycle: Before hook opens browser → steps run → After hook closes browser
- Helper functions (buildSelector, fillField, gotoUrl) simplify element interaction
- Store custom state on `this` to share data between steps in a scenario
- Extend PlaywrightWorld with custom methods for project-specific operations
- Use Cucumber profiles for different environments (local, staging, production)
- Each scenario gets complete isolation: new World, new browser, new context

Next: Writing Custom Step Definitions, creating your own Gherkin vocabulary.

Chapter 17: Writing Custom Step Definitions

webship-js gives you a comprehensive library of built-in steps: navigation, forms, assertions, waits, modals, tables, API testing. But every project has unique features that no generic library can anticipate. Maybe your app has a drag-and-drop kanban board, a custom date picker, a multi-select autocomplete, or a real-time chat widget.

This is where custom step definitions come in. You write the Gherkin that describes your feature in business language, then implement the automation behind it.

This chapter teaches you everything about writing custom step definitions: from your first simple step to advanced patterns with data tables, doc strings, and domain-specific vocabularies.

Step Definition Anatomy

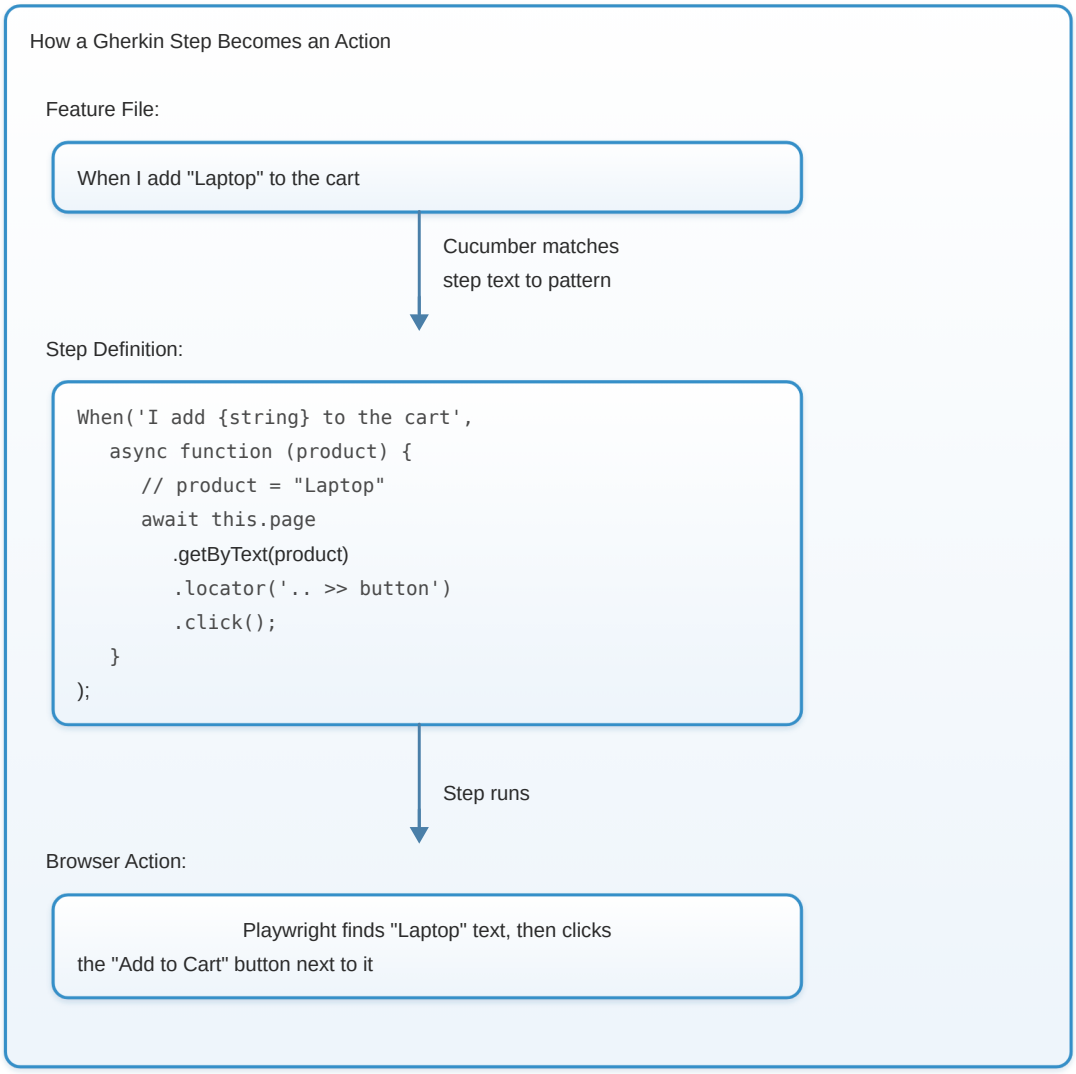


FIGURE 17.1 *Step Definition Anatomy*

Every step definition has three parts:

1. **Keyword** (`Given`, `When`, `Then`)
2. **Pattern** (string or regex that matches the Gherkin text)
3. **Function** (the code that runs when the pattern matches)

Cucumber Expressions vs Regular Expressions

webship-js uses both. Here is when to use each:

CUCUMBER EXPRESSIONS	REGULAR EXPRESSIONS
When('I add {string} to cart',	When(/^I add "([^"]*)" to
function (product) { ... }	cart$/, function (product)
);	{ ... }
	);
Simpler, more readable	More powerful, flexible
Built-in types: {string},	Full regex patterns
{int}, {float}, {word}	Capture groups
Custom parameter types	Optional parts (?:...)?
USE WHEN:	USE WHEN:
Simple parameter extraction	Complex patterns needed
New step definitions	Optional words/phrases
Readable > flexible	Matching flexibility needed

Recipe 162 Your First Custom Given Step

Problem: Your login flow is unusual, maybe it is SSO, maybe it has a custom role picker, and every scenario in the suite starts with the same six lines of clicking and filling. The day a developer renames the “Sign In” button, forty scenarios fail at the same step and the new QA hire spends an afternoon fixing forty copies of the same Gherkin. One custom Given step would have made that a one-line fix.

Solution:

```
// tests/step-definitions/custom-setup.js
const { Given } = require('@cucumber/cucumber');

// Cucumber Expression
Given('I am a logged in user with the {string} user', async function
(role) {
  const users = {
    admin: { email: 'admin@example.com', password: 'AdminPass123' },
    editor: { email: 'editor@example.com', password: 'EditorPass123' },
    customer: { email: 'customer@example.com', password: 'CustPass123' },
  };

  const user = users[role];
  if (!user) throw new Error(`Unknown user role: ${role}`);

  await this.page.goto(this.launchUrl + '/login');
  await this.page.waitForLoadState('networkidle');
  await this.page.getByLabel('Email').fill(user.email);
  await this.page.getByLabel('Password').fill(user.password);
  await this.page.getByRole('button', { name: 'Sign In' }).click();
  await this.page.waitForLoadState('networkidle');
});
```

```
Feature: Admin Dashboard
  Scenario: Admin sees dashboard
    Given I am a logged in user with the "admin" user
    When I go to "/admin/dashboard"
    Then I should see "Dashboard"
```

How It Works: Cucumber matches the Gherkin line against the Cucumber Expression `'I am a logged in user with the {string} user'`, extracts `"admin"` into the `role` parameter, and calls your function with `this` bound to the World, the same object the 411 built-in webship-js steps use. That is why `this.page` and `this.launchUrl` are already there: the built-in Before hook created the browser context and stored them before your step ran. The user map keeps credentials in one place; the `throw` on an unknown role turns a typo in a feature file into a clear error instead of a confusing login failure. Because your custom step lives in `tests/step-definitions/`, the default cucumber.js profile picks it up automatically, no registration, no imports in the feature file.

At scale: When you run twenty projects, do not hardcode the user map per project. Read it from an environment-driven JSON file (`TEST_USERS_FILE`) or environment variables, so the same step definition file can be copied, or better, shared as a small npm package, across every suite. Your CI then injects per-environment credentials as secrets, and no password ever lives in a repository. Also consider the built-in auth-state steps (save/restore storageState) so login happens once per run, not once per scenario, across 40 nightly suites that is real CI minutes saved.

Recipe 163 Your First Custom When Step

Problem: Your product pages have a card layout where the “Add to Cart” button only makes sense relative to the product it belongs to. The generic click steps can find *a* button, but not *the* button next to “Laptop Pro”, so testers start writing brittle CSS like `.product-list > div:nth-child(3) button`, and the suite breaks the first time marketing reorders the product grid. You need action steps that express the relationship: this product, its button.

Solution:

```
// tests/step-definitions/custom-actions.js
const { When } = require('@cucumber/cucumber');

When('I add {string} to the cart', async function (productName) {
  const product = this.page.locator('.product-card')
    .filter({ hasText: productName });
  await product.locator('button', { hasText: 'Add to Cart' }).click();
});

When('I remove {string} from the cart', async function (productName) {
  const item = this.page.locator('.cart-item')
    .filter({ hasText: productName });
  await item.locator('button', { hasText: 'Remove' }).click();
});

When('I set the quantity of {string} to {int}', async function
(productName, quantity) {
  const item = this.page.locator('.cart-item')
    .filter({ hasText: productName });
  await item.locator('input[type="number"]').fill(String(quantity));
});
```

```
Feature: Shopping Cart
  Scenario: Add products and update quantities
    Given I am on homepage
    When I go to "/products"
    And I add "Laptop Pro" to the cart
    And I add "Wireless Mouse" to the cart
    And I go to "/cart"
    Then I should see "Laptop Pro"
    And I should see "Wireless Mouse"
    When I set the quantity of "Laptop Pro" to 2
    And I remove "Wireless Mouse" from the cart
    Then I should see "Laptop Pro"
    And I should not see "Wireless Mouse"
```

How It Works: The pattern here is *scope, then act*. `locator('.product-card').filter({ hasText: productName })` narrows the page to the one card containing the product name, and the second `locator()` call searches only inside that card. The position of the card on the page no longer matters, reorder the grid, add a banner, paginate the list, and the step still finds the right button. Because these are `When` steps that change state, webship-js's AfterStep hook fires `smartSettle(page, 1500)` after each one: it waits for `window.__webshipAjaxCount` to hit zero, pending timers to drain, and the DOM to stay quiet for 250ms before the next step runs. That is why there is no explicit wait after the click, the cart badge update the next assertion depends on has already settled.

At scale: Notice that all three steps share the same two-line scoping idiom. When this pattern appears in your third project, extract a helper, `cartItem(page, name)`, into a shared support module so every suite scopes the same way. Teams that skip this end up with five slightly different `filter({ hasText })` variants per project, and nobody knows which one handles the edge cases.

Recipe 164 Your First Custom Then Step

Problem: "Then I should see '2'" passes when the cart badge says 2, and also when the year 2026 appears in the footer. Generic text assertions are too loose for business rules like cart counts and order totals, and when one fails, the error message ("expected to see 2") tells the reviewer nothing about *what* was wrong. The Tuesday before release, a vague failure like that costs an hour of someone re-running the scenario in headed mode just to understand the message.

Solution:

```
// tests/step-definitions/custom-assertions.js
const { Then } = require('@cucumber/cucumber');
const assert = require('assert');

Then('the cart should have {int} items', async function (expectedCount) {
  const badge = this.page.locator('.cart-badge, .cart-count').first();
  const text = await badge.textContent();
  const count = parseInt(text, 10);
  assert.strictEqual(count, expectedCount,
    `Expected ${expectedCount} items in cart, found ${count}`);
});

Then('the total should be {string}', async function (expectedTotal) {
  const total = this.page.locator('.cart-total, .order-total').first();
  const text = await total.textContent();
  assert(text.includes(expectedTotal),
    `Expected total to contain "${expectedTotal}", got "${text}"`);
});

Then('the {string} notification should appear', async function (type) {
  const selector = `.notification-${type}, .alert-${type},
.toast-${type}`;
  const notification = this.page.locator(selector).first();
  await notification.waitFor({ state: 'visible', timeout: 5000 });
});
```

```
Feature: Cart Totals
  Scenario: Cart calculates total correctly
    Given I am a logged in user with the "customer" user
    When I go to "/products"
    And I add "Laptop Pro" to the cart
    And I add "USB Cable" to the cart
    And I go to "/cart"
    Then the cart should have 2 items
    And the total should be "$1,008.99"
    And the "success" notification should appear
```

How It Works: Each assertion targets the exact element that carries the business meaning, the badge, the total, the notification, instead of searching the whole page for a string. The custom failure messages are the real payoff: `Expected 2 items in cart, found 3` tells a reviewer reading the HTML report what broke without opening a browser. The notification step uses `waitFor({ state: 'visible', timeout: 5000 })`, which polls rather than checking once, the same web-first philosophy as the built-in `Then "sel" should be visible` step (100ms polling, 5s default). One caveat: `textContent()` reads immediately, so for values that update via AJAX, lean on the AfterStep smartSettle that already ran after your last action, by the time a `Then` step executes, ajax count is zero and the DOM has been quiet for 250ms.

Pro Tip: Write the failure message before you write the assertion. If you cannot phrase what a failure should say, you have not decided what the step actually checks.

Recipe 165 Steps with String Parameters

Problem: You wrote `When I search for laptops` as a step, and now product owners want to reuse it for “wireless mouse”, “USB-C hub”, and a dozen other queries. Without parameters you would copy the step definition once per search term, the exact duplication that makes a suite rot. You need one pattern that captures whatever sits between the quotes.

Solution:

```
// Cucumber Expression: {string} captures quoted text
When('I search for {string}', async function (query) {
  await this.page.getByPlaceholder('Search').fill(query);
  await this.page.keyboard.press('Enter');
});

// Regex: ([^"]*) captures quoted text
When(/^I search for "([^"]*)"$/, async function (query) {
  await this.page.getByPlaceholder('Search').fill(query);
  await this.page.keyboard.press('Enter');
});
```

Both patterns match: `When I search for "laptops"`

How It Works: With the Cucumber Expression, `{string}` matches anything inside double quotes and hands it to your function already unquoted. The regex version does the same job with the capture group `([^"]*)`, "any run of characters that is not a quote", but you maintain the anchors (`^`, `$`) and escaping yourself. One subtle difference matters in practice: `{string}` also matches single-quoted text, so `When I search for 'laptops'` works too; the regex above only matches double quotes. Pick Cucumber Expressions for new steps and save regex for cases that genuinely need it (Recipe 170). A warning from maintaining suites across many projects: if both patterns are loaded at once, Cucumber throws an ambiguous-step error, never keep two definitions that match the same text.

Recipe 166 Steps with Integer Parameters

Problem: A search results assertion needs to work for 3 results today and 12 results after the next content import. Hardcoding the count into the step text means editing step definitions every time the data changes, and worse, someone will eventually compare the string `"12"` to the number `12` and spend twenty minutes staring at an assertion that "should obviously pass".

Solution:

```
// Cucumber Expression: {int} captures an integer
Then('I should see {int} results', async function (count) {
  const results = await this.page.locator('.search-result').count();
  assert.strictEqual(results, count);
});

When('I wait {int} seconds', async function (seconds) {
  await this.page.waitForTimeout(seconds * 1000);
});

// Regex: (\d+) captures digits
Then(/^I should see (\d+) results$/, async function (count) {
  const results = await this.page.locator('.search-result').count();
  assert.strictEqual(results, parseInt(count, 10));
});
```

Note: With regex, captured values are always strings, you need `parseInt()`. With Cucumber Expressions, `{int}` automatically converts to a number.

How It Works: `{int}` does two things: it matches a run of digits (with optional minus sign) in the Gherkin text, and it converts the match to a JavaScript number before your function sees it. That conversion is what makes `assert.strictEqual(results, count)` safe, `locator().count()` returns a number, and strict equality between a number and a string always fails. The regex version skips the conversion, which is exactly the trap the `parseInt(count, 10)` call defuses. And a word about that `I wait {int} seconds` example: it shows the mechanics of `{int}`, but treat it as a debugging tool only. webship-js has a hard rule, no static sleeps. Every built-in step settles through smartSettle (ajax count zero, timers drained, DOM quiet for 250ms), so a scenario that needs a hand-written sleep is usually telling you an assertion should be a polling web-first check instead.

Recipe 167 Steps with Data Tables

Problem: A registration form has nine fields. Writing nine `When I fill in` lines per scenario works, but the third time a product owner reviews the feature file, their eyes glaze over before line five, and when the form gains a tenth field, you edit every scenario that touches it. A table turns nine noisy lines into one readable block that a BA can review, edit, and even extend without asking engineering.

Solution:

```
const { When, Then } = require('@cucumber/cucumber');
const assert = require('assert');

// Key-value table
When('I fill in the profile form with:', async function (table) {
  for (const [field, value] of table.raw()) {
    await this.page.getByLabel(field).fill(value);
  }
});

// Multi-column table
Then('the user list should contain:', async function (table) {
  const rows = table.hashes(); // [{name: "John", role: "Admin"}, ...]
  for (const row of rows) {
    const tr = this.page.locator('tr').filter({ hasText: row.name });
    const text = await tr.textContent();
    assert(text.includes(row.role),
      `Expected "${row.name}" to have role "${row.role}"`);
  }
});
```

```
Feature: User Profile
  Scenario: Fill profile form
    When I fill in the profile form with:
      | First Name | Rajab         |
      | Last Name  | Natshah       |
      | Email      | r@example.com |
      | Phone      | +1234567890   |

  Scenario: Verify user list
    Then the user list should contain:
      | name       | role   | status |
      | John Smith | Admin  | Active |
      | Jane Doe   | Editor | Active |
```

Data Table Methods:

METHOD	RETURNS
table.raw()	[["field", "value"], ["field2", "value2"]]
table.rows()	Same as raw() but skips header row
table.rowsHash()	{"field": "value", "field2": "value2"}
table.hashes()	[{col1: "v1", col2: "v2"}, ...]
	(each row as object with header keys)

How It Works: When the last argument of a Gherkin step is a table, Cucumber passes a `DataTable` object as the final parameter of your function, after any `{string}` / `{int}` captures. The method you call decides the shape: `raw()` for key-value pairs (loop over `[field, value]`), `hashes()` for multi-column tables where the first row is treated as headers and each following row becomes an object. The profile-form step delegates each row to `getByLabel(field).fill(value)`, which is the same label-first resolution the built-in `When I fill in "Label" with "value"` step uses, the table is just a loop around the behavior testers already know. The verify step shows the matching read-side pattern: scope a `tr` by the name column, then assert the role text inside that row only.

At scale: Tables are the cheapest way to keep product owners inside the feature files. At Vardot, the registration scenarios that survived years of Varbase releases were the table-driven ones, when a field was added, a BA edited one table row in a pull request and never touched JavaScript. Also remember webship-js already ships `When I fill in the following:` as a built-in; write a custom table step only when you need behavior the built-in does not cover, like custom widgets or per-field validation checks.

Recipe 168 Steps with Doc Strings

Problem: You are testing a rich-text editor, or posting a JSON payload, and the content spans multiple paragraphs with quotes, apostrophes, and angle brackets. Squeezing that into a single quoted `{string}` parameter produces an unreadable escaping nightmare, and the first reviewer who tries to tweak the test data breaks the quoting and the whole feature file stops parsing.

Solution:

```
const { When, Then } = require('@cucumber/cucumber');

When('I fill in the editor with:', async function (content) {
  // content is the text between the triple quotes
  await this.page.locator('.rich-editor').fill(content);
});

When('I submit the API request with body:', async function (jsonString) {
  const body = JSON.parse(jsonString);
  // Use body for API request...
});
```

```
Feature: Content Editor
  Scenario: Write a long blog post
    When I fill in the editor with:
      """
      This is a long blog post about automated testing.

      It spans multiple paragraphs and includes special characters:
      quotes "like these", apostrophes it's, and <html> tags.
      """

  Scenario: Submit JSON payload
    When I submit the API request with body:
      """
      {
        "title": "New Article",
        "body": "Article content here",
        "tags": ["testing", "automation"]
      }
      """
```

How It Works: Everything between the `"""` fences arrives in your function as one plain string, last parameter, with the common leading indentation stripped, so the JSON example reaches `JSON.parse()` exactly as written, no escaping needed. Nothing inside a doc string is interpreted by Gherkin: quotes, `<html>` tags, and blank lines all pass through untouched. This is the same mechanism behind the built-in API steps (`When I send a POST request to "/path" with body:`) and the network-mock steps (`Given the URL "**/api/x" returns the JSON:`), check those before writing your own, because for API payloads and mocked responses the built-ins already parse, send, and assert for you. Reserve custom doc-string steps for content that goes *into the UI*, like the rich-editor example.

Recipe 169 Domain-Specific Steps

Problem: Here is a scene every QA lead knows: a product owner writes “when the customer places an order, inventory decreases”, QA translates it into eleven lines of clicks and fills, and three sprints later nobody, not even the author, can tell which business rule those eleven lines were protecting. The PO stops reading the feature files, sign-off becomes “I trust you”, and the suite quietly drifts away from the acceptance criteria it was supposed to encode.

Solution:

```
// tests/step-definitions/ecommerce-domain.js
const { Given, When, Then } = require('@cucumber/cucumber');
const assert = require('assert');

// Domain: E-commerce
Given('the product {string} is in stock', async function (product) {
  // Navigate and verify
  await this.page.goto(this.launchUrl + '/products');
  await this.page.waitForLoadState('networkidle');
  const row = this.page.locator('tr').filter({ hasText: product });
  const status = await row.locator('.stock-status').textContent();
  assert(status.includes('In Stock'), `${product} is not in stock`);
});

When('the customer places an order for {string}', async function (product)
{
  await this.page.goto(this.launchUrl + '/products');
  await this.page.locator('.product-card').filter({ hasText: product })
    .locator('button', { hasText: 'Add to Cart' }).click();
  await this.page.goto(this.launchUrl + '/checkout');
  await this.page.getByRole('button', { name: 'Place Order' }).click();
  await this.page.waitForLoadState('networkidle');
});

Then('the order should be confirmed', async function () {
  await this.page.locator('body').filter({ hasText: 'Order Confirmed' })
    .waitFor({ timeout: 10000 });
});

Then('the inventory for {string} should decrease by {int}', async function
(product, amount) {
  // Check inventory via API or admin panel
  await this.page.goto(this.launchUrl + '/admin/inventory');
  await this.page.waitForLoadState('networkidle');
  // Verify inventory count...
});
```

```
Feature: Order Processing
  As a customer
  I want to order products
  So that I can receive them at home

  Scenario: Successful order reduces inventory
    Given the product "Wireless Mouse" is in stock
    When the customer places an order for "Wireless Mouse"
    Then the order should be confirmed
    And the inventory for "Wireless Mouse" should decrease by 1
```

Imperative vs Declarative comparison:

```
Imperative (technical):          Declarative (business):
Given I am on homepage           Given the product "Mouse" is
When I go to "/products"           in stock
And I click "Add to Cart"        When the customer places an
  in the "Mouse" row               order for "Mouse"
And I go to "/checkout"          Then the order should be
And I fill in "Name" with "J"      confirmed
And I fill in "Address" with "1"
And I press "Place Order"
And wait for AJAX to finish
Then I should see "Confirmed"
```

Both test the same thing, but the declarative version communicates the **intent** while the imperative version communicates the **mechanism**.

How It Works: Each domain step packages a whole user journey behind one business sentence. `the customer places an order for {string}` navigates, scopes the product card by text, clicks Add to Cart, and submits checkout, five UI operations, one Gherkin line. The win is not just readability: when checkout gains a confirmation modal next quarter, you update one function instead of every scenario that orders something. The confirmation assertion uses `waitFor({ timeout: 10000 })` deliberately, order processing crosses a payment round-trip, so it deserves more than the default 5s web-first window. Keep the boundary clean: domain steps describe *what the business does*, the function body owns *how the app does it*. The moment a selector leaks into your Gherkin, you have lost the layer.

At scale: This is where multi-project teams win or lose. Define the domain vocabulary once per product domain, “places an order”, “publishes an article”, “approves the request”, and reuse the same phrasing across every project in that domain, even when the implementations differ per site. Product owners then read any project’s report and understand it, the PMO can compare quality signals across products because scenarios share a language, and a QA engineer moving between projects already knows the steps. The feature files become the contract; the step definitions absorb the per-project differences.

Recipe 170 Flexible Step Patterns

Problem: Five people write feature files on your team, and each phrases the same action a little differently: “I add 1 item”, “I add 3 items”, “the page should show”, “the screen should display”. You can either police the wording in every review, and watch undefined-step errors pile up anyway, or write patterns tolerant enough to accept the natural variations people actually type.

Solution:

```
// Regex with optional parts
When(/^I (add|remove) (\d+) items? (to|from) the cart$/, async function
(action, count, preposition) {
  // Matches:
  //   "I add 1 item to the cart"
  //   "I add 3 items to the cart"
  //   "I remove 1 item from the cart"
  //   "I remove 5 items from the cart"
});

// Optional pronoun
When(/^(?:I |we )?click the submit button$/, async function () {
  // Matches: "I click the submit button" AND "click the submit button"
});

// Alternative words
Then(/^the (page|screen|view) should (show|display|contain) "([^"]*)"$/,
async function (container, verb, text) {
  // Matches: "the page should show 'Hello'"
  //          "the screen should display 'Hello'"
  //          "the view should contain 'Hello'"
});
```

How It Works: This is the territory where regex earns its keep over Cucumber Expressions. `items?` makes the trailing “s” optional, so singular and plural both match. `(?:I |we )?` is a *non-capturing* optional group, the `?:` prefix means it does not produce a function parameter, which keeps your argument list aligned with the captures you actually care about. Alternations like `(add|remove)` capture which word matched, so one function can branch on the action instead of two near-identical definitions drifting apart. Always anchor with `^` and `$`, an unanchored pattern can match in the middle of a longer step and collide with another definition, and Cucumber’s ambiguous-step error on a Friday afternoon is nobody’s favorite.

Pro Tip: Flexibility has a budget. Two or three tolerant patterns per domain keep authors happy; a regex with six alternations becomes a step nobody can discover or document. webship-js’s own rule is a good one to copy: every step gets at least five documented example lines, so the accepted phrasings are written down where authors can find them.

At scale: Across many projects, prefer *one* published phrasing per action and use flexible patterns only as a compatibility net for the variants that already exist in older suites. A shared step-reference page (or the built-in step inventory) beats clever regex, new hires learn the canonical wording once and carry it to every project they touch.

Step Definition Organization

```
tests/
├── step-definitions/
│   ├── webship.js              ◀ Built-in (from webship-js)
│   ├── webship-api.js          ◀ Built-in API (from webship-js)
│   ├── custom-auth.js          ◀ Your login/logout steps
│   ├── custom-cart.js          ◀ Your e-commerce steps
│   ├── custom-admin.js         ◀ Your admin panel steps
│   ├── custom-viewport.js      ◀ Your responsive testing steps
│   └── custom-a11y.js          ◀ Your accessibility steps
├── features/
│   ├── auth/
│   │   ├── login.feature
│   │   └── registration.feature
│   ├── products/
│   │   ├── listing.feature
│   │   └── cart.feature
│   └── admin/
│       ├── dashboard.feature
│       └── users.feature
└── support/
    └── hooks.js                ◀ Custom hooks
```

FIGURE 17.2 *Step Definition Organization*

Chapter Summary

- Step definitions connect Gherkin text to Playwright automation code
- Three parts: keyword (Given/When/Then), pattern (expression/regex), function (async code)
- Cucumber Expressions are simpler; Regular Expressions are more flexible
- `{string}` captures quoted text, `{int}` captures numbers
- Data tables pass structured key-value or multi-column data
- Doc strings pass multiline text (JSON, HTML, long content)
- Domain-specific steps make Gherkin read like business language
- Flexible patterns handle pluralization and wording variations
- Organize step definitions by domain (auth, cart, admin) not by keyword
- All steps share the same World (`this`) within a scenario

Next: Hooks and Lifecycle, controlling what happens before, during, and after your tests.

Chapter 18: Hooks and Lifecycle

Hooks are the invisible machinery that runs before and after your scenarios, steps, and the entire test suite. They set up the browser, clean up after failures, capture screenshots, seed test data, and tear down environments. Without hooks, you would need to repeat the same setup and cleanup code in every scenario.

This chapter covers every hook type and shows you how to use them effectively.

Hook Execution Order

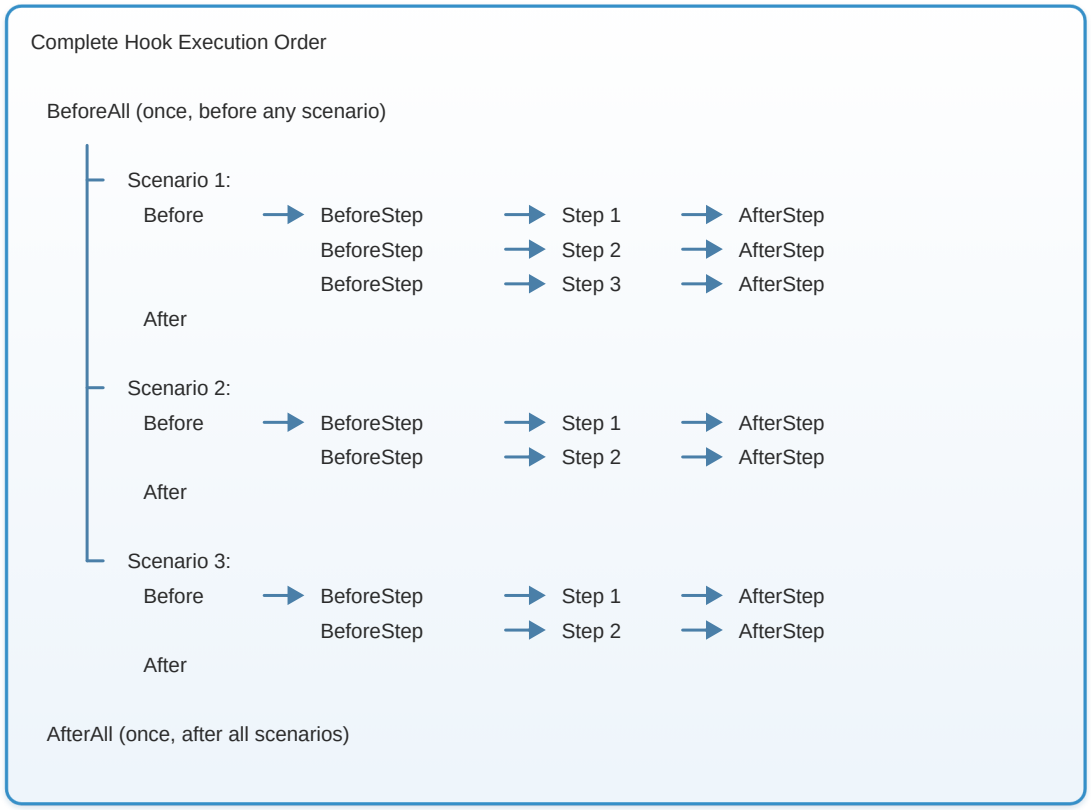


FIGURE 18.1 *Hook Execution Order*

Recipe 171 Before Hook, Browser Setup

Problem: A new team member clones the repo, runs the suite, and asks the question everyone asks in their first week: "where does `this.page` come from? Nobody opens a browser in any scenario." If you cannot answer that, you cannot debug the day the browser *fails* to open, a bad `BROWSER` env value, a missing Playwright binary in CI, a context that leaks state between scenarios. The setup is built in, but you still need to understand it.

Solution:

This is already built into webship-js:

```
const { Before } = require('@cucumber/cucumber');

Before(async function () {
  await this.openBrowser();
  if (this.minWaitTime.before_scenario > 0) {
    await this.page.waitForTimeout(this.minWaitTime.before_scenario);
  }
});
```

This runs before every scenario:

1. Opens a new browser instance
2. Creates a new browser context (like incognito mode)
3. Creates a new page/tab
4. Optionally waits the configured `before_scenario` time

How It Works: The hook reads its launch options from `playwright.config.ts`: `BROWSER` picks chromium, firefox, or webkit; `HEADLESS=false` gives you a visible window; `SLOW_MO` throttles every action (default 0 in CI, 300ms headless, 800ms headed, slow enough to watch). The fresh context per scenario is the part worth internalizing: cookies, localStorage, and sessions from the previous scenario are gone, which is what makes scenario order irrelevant. This is also where webship-js plants its smart-wait instrumentation, `addInitScript` installs `window.__webshipAjaxCount`, `window.__webshipPendingTimers`, and `window.__webshipLastMutation` into every page before your app's first line of JavaScript runs. Every smartSettle call for the rest of the scenario reads those three globals.

At scale: Resist the urge to copy this hook into your project and modify it, that fork is how twenty projects end up on twenty slightly different lifecycles. Configure instead: `BROWSER`, `HEADLESS`, and `LAUNCH_URL` as environment variables in CI mean the identical test code runs against chromium on the staging URL tonight and webkit on production tomorrow, with the pipeline, not the repo, deciding which.

Recipe 172 After Hook, Browser Cleanup

Problem: Your nightly run dies halfway through with an out-of-memory error, and when DevOps checks the runner there are sixty zombie Chromium processes eating 12 GB of RAM. That is what happens when cleanup only runs on success, every failed scenario leaves a browser behind, and on a bad night the failures are exactly when you have the most of them.

Solution:

Built into webship-js:

```
const { After } = require('@cucumber/cucumber');

After(async function () {
  if (this.minWaitTime.after_scenario > 0) {
    await this.page.waitForTimeout(this.minWaitTime.after_scenario);
  }
  await this.closeBrowser();
});
```

The After hook ALWAYS runs, even if the scenario fails. This prevents browser instances from accumulating.

How It Works: Cucumber treats After hooks like a `finally` block: a step can throw, an assertion can fail, the scenario can be marked failed, the After hook still executes. `closeBrowser()` tears down page, context, and browser in order, releasing the OS process. This ordering also explains why failure screenshots (Recipe 173) must be captured in an After hook *before* this one runs, once the browser closes, the page you want to photograph no longer exists. Remember the reverse-execution rule from the end of this chapter: After hooks run last-defined-first, and webship-js defines its cleanup hook early, so your custom After hooks get the still-open browser.

At scale: On shared CI runners, leaked browsers are not just your problem, they starve the next project's job on the same machine. If you ever bypass the built-in lifecycle (custom worlds, experimental hooks), add a belt-and-suspenders `pkill -f chromium` or a container-per-job policy in the pipeline. A container that dies with the job is the only cleanup that never has a bug.

Recipe 173 Screenshot on Failure

Problem: The nightly run failed at 2 a.m., the morning standup wants to know if the release is safe, and all the log says is “expected ‘Order Confirmed’ to be visible”. Was it a broken deploy? A cookie banner covering the button? A half-rendered page? Without a picture of the moment of failure, someone has to reproduce the run locally, and the failures that matter most are the ones that only happen in CI.

Solution:

```
const { After, Status } = require('@cucumber/cucumber');
const path = require('path');
const fs = require('fs');

After(async function (scenario) {
  if (scenario.result.status === Status.FAILED && this.page) {
    const dir = path.join(process.cwd(), 'screenshots');
    if (!fs.existsSync(dir)) fs.mkdirSync(dir, { recursive: true });

    const name = scenario.pickle.name.replace(/[^a-zA-Z0-9]/g, '-
').toLowerCase();
    const timestamp = Date.now();

    // Save to file
    await this.page.screenshot({
      path: path.join(dir, `${name}-${timestamp}.png`),
      fullPage: true,
    });

    // Attach to Cucumber report
    const buffer = await this.page.screenshot({ fullPage: true });
    this.attach(buffer, 'image/png');
  }
});
```

How It Works:

```
Scenario result:
  PASSED → After hook does nothing special → close browser
  FAILED → After hook captures screenshot → attach to report → close
browser

The scenario.result.status can be:
  Status.PASSED    → All steps succeeded
  Status.FAILED    → A step threw an error or assertion failed
  Status.PENDING   → A step is marked as pending
  Status.UNDEFINED → A step has no matching definition
  Status.SKIPPED   → A step was skipped (after a failure)
```

The `this.attach(buffer, 'image/png')` call is the half people forget: it embeds the image into the Cucumber JSON report, so the generated HTML report shows the failure picture inline, right under the red step. A screenshot file on a CI runner's disk disappears with the runner; a screenshot inside the report travels with it.

Pro Tip: webship-js already does this out of the box, `WEBSHIP_SCREENSHOT_ON_FAILED` defaults to true, failures land in `WEBSHIP_SCREENSHOT_DIR` (default `./screenshots`) with the `failed_` prefix, and the filename pattern (`{datetime}.{feature_file}.feature_{step_line}.{ext}`) encodes which feature and line failed. Write a custom hook like this only when you need something the built-in does not give you, such as attaching extra context or pushing the image to external storage.

At scale: Filenames are metadata. When forty suites upload screenshots as CI artifacts, `failed_2026-06-10.checkout.feature_42.png` is searchable and groupable; `screenshot-1718000000.png` is noise. Keep the webship-js pattern (or extend it with `WEBSHIP_SCREENSHOT_PATTERN`), have CI upload the screenshots directory plus the HTML report on failure only, and your delivery managers can answer "what exactly broke on project X last night" from the artifact browser, no engineer required.

Recipe 174 BeforeStep / AfterStep Hooks

Problem: You are watching a headed run to figure out why a scenario misbehaves, but the steps fly past faster than your eyes can follow. Or the opposite: a scenario fails in CI and the log gives you a wall of Gherkin with no indication of *which* step was running when things went sideways. Per-step hooks are the lever for both, slow it down when a human is watching, narrate it when a machine is.

Solution:

Built into webship-js:

```
const { BeforeStep, AfterStep } = require('@cucumber/cucumber');

BeforeStep(async function () {
  if (this.page && this.minWaitTime.before_step > 0) {
    await this.page.waitForTimeout(this.minWaitTime.before_step);
  }
});

AfterStep(async function () {
  if (this.page && this.minWaitTime.after_step > 0) {
    await this.page.waitForTimeout(this.minWaitTime.after_step);
  }
});
```

Custom AfterStep: Log every step:

```
AfterStep(async function (step) {
  const text = step.pickle?.step?.text || step.pickleStep?.text || '';
  const status = step.result?.status || 'unknown';
  console.log(`  [${status}] ${text}`);
});
```

How It Works: `BeforeStep` and `AfterStep` wrap every single step in every scenario, with full World access. The most important AfterStep hook in webship-js is one you never see: after every state-changing step, it calls `smartSettle(page, 1500)`, waiting for in-flight fetch/XHR to

reach zero, pending timers to drain, and the DOM to stay quiet for at least 250ms, polled every 50ms. That hidden hook is the reason webship-js suites do not need “wait for AJAX” steps sprinkled everywhere, and the reason the `minWaitTime` delays above should stay at zero in normal runs: smartSettle waits for *events*, a delay waits for *time*, and the repo rule is wait for events, not time. The logging variant costs nothing and pays for itself the first time a CI log shows you exactly which step was last alive. Note the defensive `step.pickle?.step?.text || step.pickleStep?.text`, the hook payload shape has shifted between Cucumber versions, and this handles both.

At scale: Keep step delays out of committed config. If one slow legacy environment needs breathing room, gate it behind an environment variable in that project’s pipeline rather than slowing all 40 suites, a 500ms per-step delay across a 2,000-step nightly farm is over 16 minutes of paid CI time spent sleeping.

Recipe 175 Tagged Hooks (Conditional Setup)

Problem: Five of your eighty scenarios need a seeded admin dataset. Put the seeding in a global Before hook and every scenario pays the cost, the suite slows down for everyone because of five scenarios. Put it inside each scenario as Given steps and you have five copies to keep in sync. The third option is the right one: setup that attaches itself to a tag, so scenarios opt in with one word.

Solution:

```
const { Before, After } = require('@cucumber/cucumber');

// Only runs before scenarios tagged @admin
Before({ tags: '@admin' }, async function () {
  // Seed admin-specific test data
  await this.page.goto(this.launchUrl + '/api/test/seed-admin-data');
});

// Only runs before scenarios tagged @email
Before({ tags: '@email' }, async function () {
  // Clear the test email inbox
  await this.page.goto(this.launchUrl + '/api/test/clear-emails');
});

// Only runs after scenarios tagged @cleanup
After({ tags: '@cleanup' }, async function () {
  // Clean up test data
  await this.page.goto(this.launchUrl + '/api/test/cleanup');
});

// Runs for scenarios with BOTH tags
Before({ tags: '@admin and @staging' }, async function () {
  // Special staging admin setup
});

// Runs for scenarios with EITHER tag
Before({ tags: '@smoke or @regression' }, async function () {
  // Common setup for test suites
});
```

```
@admin
Feature: Admin Management
  @cleanup
  Scenario: Admin creates and deletes a user
    Given I am a logged in user with the "admin" user
    When I go to "/admin/users"
    And I press "Add User"
    ...

@email
Feature: Email Notifications
  Scenario: User receives welcome email
    Given I am on homepage
    When I go to "/register"
    ...
```

How It Works:

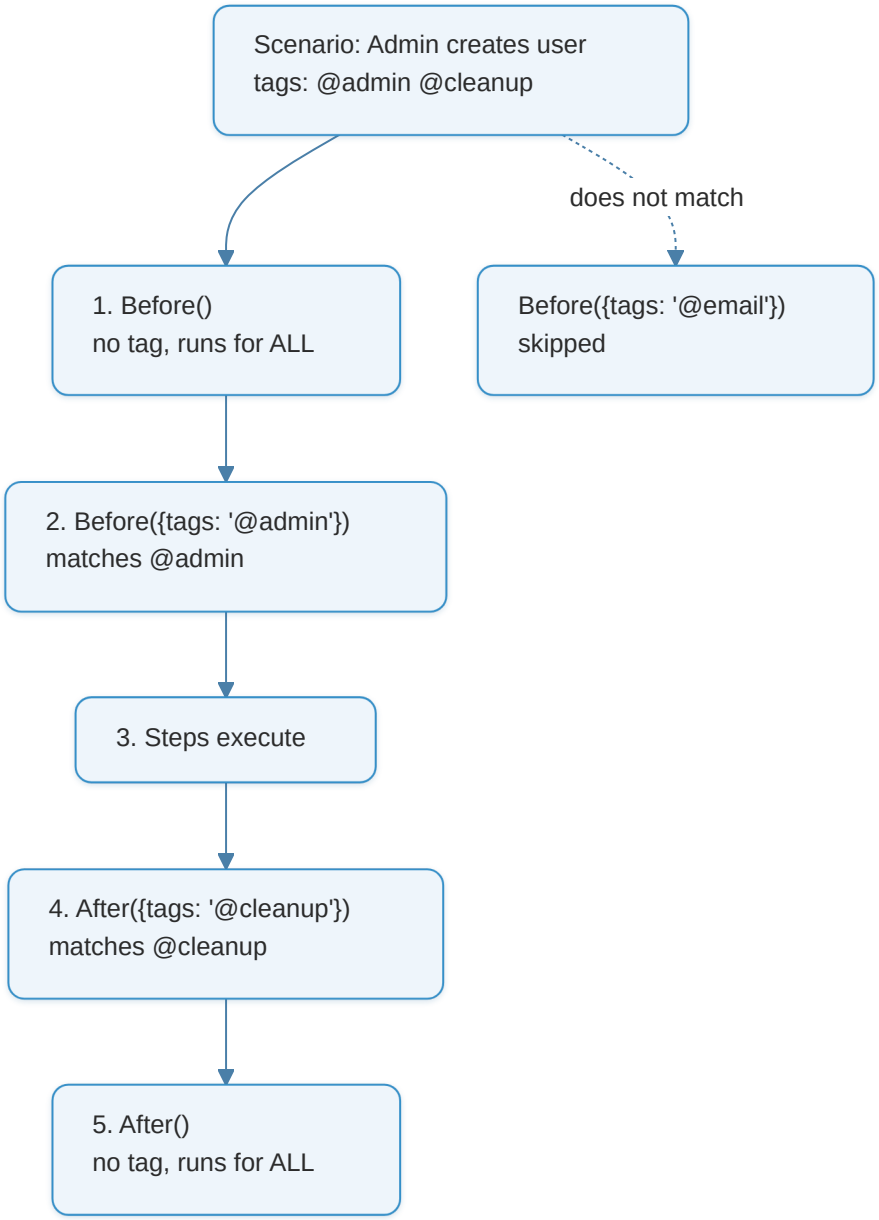


FIGURE 18.2 *Tagged Hooks (Conditional Setup) (Recipe 175)*

Tag expressions support `and`, `or`, and `not`, and tags inherit downward, a tag on the `Feature:` line applies to every scenario in the file, which is why `@admin` above covers both scenarios without repeating it. This is the same tag machinery webship-js's own behavior switches ride on: `@screenshots` turns on every-step capture, `@video` forces recording, `@js-fail` promotes console errors to failures. Your custom tagged hooks and the built-in ones coexist on the same scenario.

At scale: Tagged hooks only stay manageable if the tag vocabulary is shared. Agree on one list across all projects, `@smoke`, `@critical`, `@db-cleanup`, `@email` mean the same thing everywhere, and document it in one place. The payoff is compound: CI pipelines can run `--tags @smoke` identically on every project, dashboards can compare `@critical` pass rates across products, and an engineer rotating onto a new project already knows what every tag does. The failure mode is real too: we have seen teams where `@slow` triggered cleanup on one project and skipped CI on another. Nobody trusted tags after that.

Recipe 176 BeforeAll / AfterAll (Global Setup/Teardown)

Problem: Your mock server takes eight seconds to boot. Started per scenario, that is eight seconds times two hundred scenarios, nearly half an hour of pure startup. Some work belongs to the *run*, not to any scenario: booting services, seeding shared fixtures, warming the application cache, and, at the other end, tearing it all down and announcing the result.

Solution:

```
const { BeforeAll, AfterAll } = require('@cucumber/cucumber');

BeforeAll(async function () {
  console.log('Test suite starting...');
  // Start a test database
  // Seed global fixtures
  // Start a mock server
  // Warm up the application
});

AfterAll(async function () {
  console.log('Test suite complete.');
  // Generate HTML report
  // Clean up test database
  // Stop mock server
  // Send notification to Slack
});
```

Important: `BeforeAll` and `AfterAll` do NOT have access to the World object (`this` is not available). They run in a global context, not per-scenario. Use them for infrastructure setup, not for browser operations.

BeforeAll vs Before:

BeforeAll: Runs ONCE before all scenarios
- No World access (no this.page, this.launchUrl)
- Global setup: databases, servers, reports
- Cannot interact with the browser

Before: Runs ONCE per scenario
- HAS World access (this.page, this.launchUrl)
- Scenario setup: open browser, seed data, log in
- Can interact with the browser

FIGURE 18.3 *BeforeAll / AfterAll (Global Setup/Teardown) (Recipe 176)*

How It Works: The "no World" rule is not arbitrary, the World is created fresh per scenario, and `BeforeAll` runs before the first one exists. So `this.page` is undefined there by design. Share data from `BeforeAll` through module-level variables or environment variables, never through the World. Two more traps worth knowing: when you run with `--parallel`, each worker process runs its *own* `BeforeAll` / `AfterAll`, so "once" really means "once per worker", global setup that must truly happen once belongs in the CI pipeline, before Cucumber starts. And an `AfterAll` is skipped if the process is killed mid-run (timeout, runner eviction), so anything that *must* be cleaned up needs a pipeline-level fallback too. Report generation, by the way, is already handled: webship-js writes the Cucumber JSON to `tests/reports/cucumber_report.json` and `bin/generate-reports.js` turns it into HTML and PDF, no AfterAll needed for that.

Recipe 177 Database Cleanup in After Hook

Problem: Monday's run creates `test-user@example.com`. Tuesday's run tries to create the same user, gets "email already exists", and fails, not because the app broke, but because Monday's leftovers are still in the database. Leftover test data is one of the most common sources of "flaky" failures that are not flaky at all: the test is deterministic, the *environment* is not. Cleanup has to run even when the scenario fails, which is exactly what After hooks guarantee.

Solution:

```
const { After } = require('@cucumber/cucumber');
const axios = require('axios');

After({ tags: '@db-cleanup' }, async function () {
  // Call a test-only API endpoint that resets the database
  await axios.post(this.launchUrl + '/api/test/reset-db', {}, {
    headers: { 'X-Test-Key': 'test-secret-key' },
  });
});
```

```
@db-cleanup
Feature: User Management
  Scenario: Create a user
    Given I am a logged in user with the "admin" user
    When I go to "/admin/users"
    And I press "Add User"
    And I fill in "Email" with "test-user@example.com"
    And I press "Save"
    And wait for AJAX to finish
    Then I should see "test-user@example.com"
    # After hook will clean up the created user
```

How It Works: The hook calls a test-only reset endpoint with axios, the same HTTP client behind webship-js's built-in API steps, instead of clicking through admin delete screens, because cleanup through the UI is slow and can itself fail. Because the hook is tagged `@db-cleanup`, only scenarios that create data pay the reset cost. Three rules keep this safe. First, the endpoint must exist only

in test environments, guard it behind the `X-Test-Key` header *and* an environment check on the server, because a reset endpoint reachable in production is a career-defining incident. Second, make the reset idempotent: it must succeed whether the scenario created one record or crashed before creating any. Third, prefer deleting by a recognizable marker (everything matching `test-*@example.com`) over truncating tables, so two suites sharing a staging database cannot wipe each other's fixtures. One more note on the scenario itself: the explicit AJAX wait is belt-and-suspenders here, in webship-js 2.0 the AfterStep hook already settles after "Save" via smartSettle, so the assertion never races the request.

At scale: Per-scenario cleanup keeps shared environments healthy, but defense in depth wins: pair it with a nightly environment reset job owned by DevOps, so a crashed runner that skipped its After hooks cannot poison Wednesday's runs for every project pointed at that database. And standardize the test-data marker convention (`test-` prefix, reserved email domain) across all projects, it makes the nightly sweep one query instead of forty.

Hook Priority and Ordering

When you have multiple hooks of the same type, they run in the order they are defined:

```
// These run in order: first defined = first executed
Before(async function () {
  console.log('Hook 1: Open browser');
  await this.openBrowser();
});

Before(async function () {
  console.log('Hook 2: Set cookie consent');
  await this.context.addCookies([{
    name: 'cookie_consent', value: 'accepted',
    domain: 'localhost', path: '/',
  }]);
});

// After hooks run in REVERSE order: last defined = first executed
After(async function () {
  console.log('Hook A: Close browser');
  await this.closeBrowser();
});

After(async function () {
  console.log('Hook B: Capture screenshot');
  // This runs BEFORE Hook A
});
```

```
Before hooks: defined order (1 → 2 → 3)
After hooks:  reverse order (3 → 2 → 1)
```

Chapter Summary

- Hooks run code before/after scenarios, steps, and the test suite
- `Before` / `After` run per scenario with World access
- `BeforeStep` / `AfterStep` run per step (useful for delays, logging)
- `BeforeAll` / `AfterAll` run once for the entire suite (no World access)
- Tagged hooks run only for scenarios with matching tags
- After hooks always run, even if the scenario fails (critical for cleanup)
- Screenshot on failure uses `After` with `Status.FAILED` check
- Database cleanup should use tagged After hooks
- Multiple Before hooks run in definition order; After hooks run in reverse

Next: Organizing a Large Test Suite, file structure, tag strategies, and scaling.

Chapter 19: Organizing a Large Test Suite

A test suite with five features is easy to manage. Put them all in one folder, run them all at once, done. But a mature application might have fifty features, three hundred scenarios, and twenty custom step definition files. Without a clear organizational strategy, you end up with a tangled mess where no one can find anything, tests run for hours, and new team members are afraid to add tests because they might break something.

This chapter shows you how to organize, tag, and scale a large webship-js test suite.

File Organization Patterns

Recipe 178 Domain-Based Directory Structure

Problem: It starts innocently: five feature files in one folder. A year later there are sixty, named things like `test2.feature` and `new-checkout-final.feature`, and the new hire's first PR adds a login scenario to a file that already had three, because nobody could tell where login tests

live. When a product owner asks “do we test password reset?”, the honest answer is twenty minutes of grep. Structure is not bureaucracy; it is the difference between a suite people extend and a suite people fear.

Solution:

```
tests/
├── features/                          ◀ Feature files by domain
│   ├── auth/
│   │   ├── login.feature
│   │   ├── registration.feature
│   │   ├── password-reset.feature
│   │   └── logout.feature
│   ├── products/
│   │   ├── listing.feature
│   │   ├── search.feature
│   │   ├── cart.feature
│   │   └── checkout.feature
│   ├── admin/
│   │   ├── dashboard.feature
│   │   ├── users.feature
│   │   ├── content.feature
│   │   └── settings.feature
│   ├── api/
│   │   ├── users-api.feature
│   │   ├── products-api.feature
│   │   └── orders-api.feature
│   └── common/
│       ├── navigation.feature
│       ├── error-pages.feature
│       └── accessibility.feature
│
├── step-definitions/                  ◀ Step definitions by domain
│   ├── webship.js                     ◀ Built-in (from webship-js)
│   ├── webship-api.js                 ◀ Built-in API (from webship-js)
│   ├── auth-steps.js                  ◀ Login, logout, roles
│   ├── product-steps.js               ◀ Cart, checkout, inventory
│   ├── admin-steps.js                 ◀ Admin panel operations
│   ├── viewport-steps.js              ◀ Responsive testing
│   └── a11y-steps.js                  ◀ Accessibility checks
│
├── support/                           ◀ Support files
│   ├── hooks.js                       ◀ Custom hooks
│   └── helpers.js                     ◀ Shared helper functions
│
└── assets/                            ◀ Test data files
    ├── test-image.png
    ├── test-document.pdf
    └── sample-csv.csv
```

FIGURE 19.1 *Domain-Based Directory Structure (Recipe 178)*

Rules:

- Feature files are organized by **business domain** (auth, products, admin)
- Step definitions are organized by **domain**, not by keyword (Given/When/Then)
- Support files contain hooks and shared utilities
- Assets folder holds files used in upload tests

How It Works: The default cucumber.js profile loads every step file matching `tests/step-definitions/**/*.js` and every feature matching `tests/features/**/*.feature`, recursively. That `**` is what makes this structure free: you can nest folders as deep as the domains demand and never touch configuration. The deeper win is that the directory tree doubles as a coverage map. `ls tests/features/auth/` answers "what do we test about authentication?" without opening a single file, and `npx cucumber-js tests/features/auth/` runs exactly that answer. Add `tests/selectors/` to the picture too, named-selector JSON files auto-load from there via `worldParameters.selectors.filesPath`, and a stranger can navigate the whole suite from the folder names alone.

At scale: Make this tree *identical* across every project you maintain. When all twenty suites share the same skeleton, same folder names, same domains where they apply, a QA engineer rotating from project A to project B knows where everything lives before opening the repo, and your scaffolding scripts, CI templates, and onboarding docs are written once. At Vardot this was the quiet rule that made Varbase testing survivable: the structure was boring on purpose, and boring scales.

Recipe 179 Step Definition Organization

Problem: Someone needs to fix the logout step. They open `when-steps.js`, 300 lines of unrelated actions sorted by keyword, and scroll past cart steps, admin steps, and viewport steps hunting for it. Worse, the matching assertions live in a different file, so understanding "how do we test auth?" means reading three files in parallel. Keyword-sorted step files are the junk drawer of test suites: everything is technically somewhere.

Solution:

Split step definitions by domain, keeping related steps together:

```
// tests/step-definitions/auth-steps.js
// All authentication-related steps in one file

const { Given, When, Then } = require('@cucumber/cucumber');

// Users configuration
const users = {
  admin: { email: 'admin@example.com', password: 'AdminPass123' },
  editor: { email: 'editor@example.com', password: 'EditorPass123' },
  customer: { email: 'customer@example.com', password: 'CustPass123' },
};

Given('I am a logged in user with the {string} user', async function
(role) {
  const user = users[role];
  await this.page.goto(this.launchUrl + '/login');
  await this.page.waitForLoadState('networkidle');
  await this.page.getByLabel('Email').fill(user.email);
  await this.page.getByLabel('Password').fill(user.password);
  await this.page.getByRole('button', { name: 'Sign In' }).click();
  await this.page.waitForLoadState('networkidle');
});

When('I log out', async function () {
  await this.page.locator('a, button').filter({ hasText: 'Logout'
}).first().click();
  await this.page.waitForLoadState('networkidle');
});
```

```
// tests/step-definitions/product-steps.js
// All product/cart-related steps

const { When, Then } = require('@cucumber/cucumber');
const assert = require('assert');

When('I add {string} to the cart', async function (product) { ... });
When('I remove {string} from the cart', async function (product) { ... });
Then('the cart should have {int} items', async function (count) { ... });
Then('the total should be {string}', async function (total) { ... });
```

Avoid organizing by keyword (all Given steps in one file, all When steps in another). That makes it impossible to understand a domain at a glance:

```
BAD:                                GOOD:
given-steps.js  ← 200 lines     auth-steps.js      ← focused
when-steps.js   ← 300 lines     product-steps.js   ← focused
then-steps.js   ← 200 lines     admin-steps.js     ← focused
```

How It Works: Cucumber does not care how you split the files, all step definitions land in one global registry at load time, and any feature can use any step. The organization is purely for humans, which is exactly why it matters: a domain file is a self-contained story (here is how we log in, here is how we log out, here is the user data both share), while a keyword file is a concordance. Keeping the `users` map at the top of `auth-steps.js` is the same principle at file level, the data lives next to the only steps that use it. One rule the global registry enforces for you: a step pattern must be unique across *all* files, so a duplicate `When I log out` in two domain files is a load-time ambiguity error, not a silent override. Take it as a feature, it forces the conversation about which domain owns the step. webship-js follows this layout itself: 411 steps across 36 domain-focused files (forms, navigation, api, cookies, clock, iframe…), not three keyword buckets.

At scale: The domain split is also your extraction roadmap. When `auth-steps.js` looks the same in your fourth project, that file wants to become a shared internal package, one version, one fix, twenty consumers. Files organized by keyword never reveal this; files organized by domain make the duplication impossible to miss.

Tag Strategies

Recipe 180 Tag Naming Conventions

Problem: Six months in, your features carry `@smoke`, `@Smoke`, `@quick`, `@fast`, and `@critical-path`, five tags, one meaning, invented by five people who never agreed on a vocabulary. Then the release pipeline runs `--tags "@smoke"` and silently skips the scenarios tagged `@Smoke`, and nobody notices until a broken login ships. Tags are an API between your features and your pipelines; an API needs a published contract.

Solution:

```
# Severity tags
@smoke           # Critical path tests (run on every PR)
@regression      # Full regression suite (run nightly)
@full            # Everything including edge cases

# Feature area tags
@auth            # Authentication features
@products        # Product-related features
@admin           # Admin panel features
@api             # API tests (no browser)

# Environment tags
@local           # Only run locally
@staging         # Only run on staging
@production      # Safe for production

# Status tags
@wip             # Work in progress (skip in CI)
@skip            # Temporarily skipped (with reason in description)
@flaky           # Known flaky test (quarantined)

# Browser tags
@chromium        # Chromium-specific
@skip-webkit     # Skip on WebKit

# Type tags
@a11y            # Accessibility tests
@visual          # Visual regression tests
@performance     # Performance-related tests
```

How It Works: Tags are case-sensitive strings matched literally by Cucumber's tag-expression engine, there is no fuzzy matching and no warning for a tag that nothing selects, which is why the convention has to live in a document, not in tribal memory. Keep the dimensions orthogonal: one severity tag, one feature-area tag, plus status tags as needed, so `@smoke @auth` composes cleanly in expressions like `--tags "@smoke and @auth"`. Reserve a namespace for the tags webship-js itself interprets, `@screenshots`, `@video`, `@no-video`, `@js-fail`, `@js-warn`, `@js-off`,

these are not labels, they change runtime behavior, and a teammate inventing `@video` to mean "tests the video player" will have a confusing afternoon. And every `@skip` or `@flaky` should carry a reason and a ticket reference in the scenario description; a quarantine tag without an owner is how flaky tests become permanent residents.

At scale: This is the single highest-leverage standardization for a multi-project team. One tag vocabulary, versioned in a shared document, adopted by every suite, because your CI templates hard-code `--tags "@smoke"`, your dashboards group by `@critical` pass rate per project, and your PMO reads those dashboards to answer "is release X safe?". A project that renames `@smoke` to `@quick` does not just break its own pipeline; it falls off every cross-project report.

Recipe 181 Test Suite Tiers

Problem: Your full suite takes 45 minutes. Run it on every pull request and developers stop waiting for green, they merge on faith, or they stop writing tests because tests are "what makes CI slow". Run only a hand-picked few and a regression slips through the gap. The way out is not one suite but a ladder: a two-minute answer for every PR, a deeper answer every night, and the full answer before a release.

Solution:

```
# Tier 1: Smoke Tests (run on every PR, ~2 minutes)
npx cucumber-js --tags "@smoke"

# Tier 2: Regression (run nightly, ~15 minutes)
npx cucumber-js --tags "@smoke or @regression"

# Tier 3: Full Suite (run on release, ~45 minutes)
npx cucumber-js --tags "not @skip and not @wip"

# Run specific domain
npx cucumber-js --tags "@auth"
npx cucumber-js --tags "@api"

# Exclude flaky tests
npx cucumber-js --tags "not @flaky"

# Complex tag expressions
npx cucumber-js --tags "@smoke and not @flaky"
npx cucumber-js --tags "(@auth or @products) and not @wip"
```

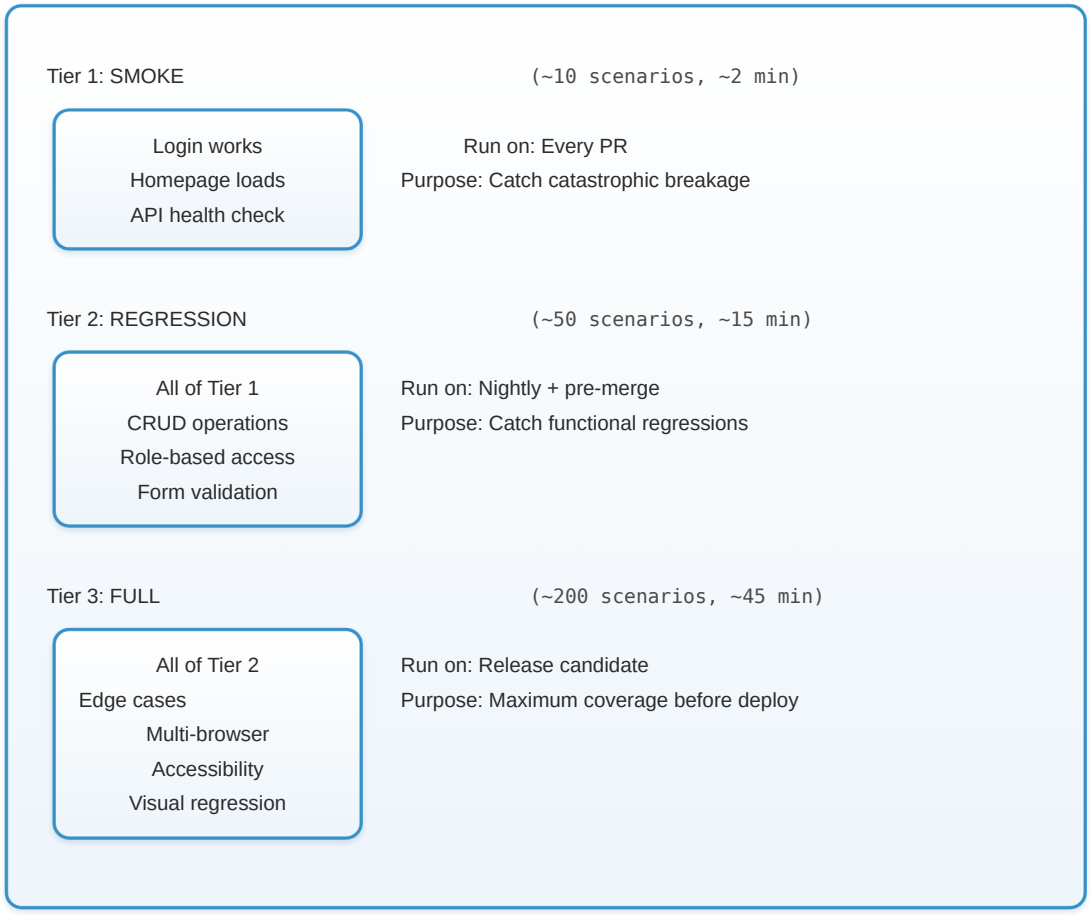


FIGURE 19.2 *Test Suite Tiers (Recipe 181)*

How It Works: The tiers are nothing but tag expressions, the same suite, filtered three ways. `@smoke or @regression` makes the tiers cumulative by construction, so a scenario promoted to smoke is never accidentally dropped from the nightly. `not @skip and not @wip` for the release tier flips the logic: instead of opting scenarios *in*, it runs everything that has not been explicitly benched, which is the right default when the stakes are highest. Discipline lives at the boundaries. Smoke membership should be a review decision with a budget, “smoke stays under three minutes” is a rule someone has to enforce, because every author believes their scenario is critical, and a fifteen-minute smoke tier is just a slow regression tier with a misleading name. Each tier writes the same `tests/reports/cucumber_report.json`, so one report pipeline serves all three.

At scale: Standardize the tier *names and meanings* across projects, not the scenario counts, a small brochure site's regression tier might be eight scenarios, a commerce platform's might be three hundred, and that is fine. What must match is the contract: every project answers "smoke" in minutes on every PR and "full" before each release. That symmetry lets DevOps build one pipeline template with the tag expression as a parameter, and lets the PMO read "smoke green, regression green" identically across the whole portfolio. It is also your CI budget knob: nightly cost scales with the regression tier, so when the farm bill climbs, you tune tags rather than re-architect pipelines.

Recipe 182 Environment-Specific Tags

Problem: A test that resets the database is a feature on localhost and a disaster on production. A payment-gateway test only makes sense where the sandbox gateway is wired up. The night someone points the full suite at the wrong base URL, and someone eventually will, the only thing standing between "oops" and "incident report" is whether your scenarios know which environments they are allowed to touch.

Solution:

```
@local
Feature: Development-Only Tests
  Scenario: Debug endpoint returns data
    Given the API base URL is "http://localhost:8080"
    When I send a GET request to "/api/debug/info"
    Then the API response code should be 200

@staging
Feature: Staging Integration
  Scenario: Third-party payment gateway works
    Given I am on "https://staging.example.com"
    ...

@production @readonly
Feature: Production Health Check
  Scenario: Homepage loads
    Given I am on "https://www.example.com"
    Then I should see "Welcome"
```

```
# Local development
npx cucumber-js --tags "@local or (not @staging and not @production)"

# Staging deployment
npx cucumber-js --tags "@staging or @smoke"

# Production health check (read-only tests only)
npx cucumber-js --tags "@production and @readonly"
```

How It Works: Two mechanisms cooperate here. The tag expression decides *which scenarios run*; the `LAUNCH_URL` environment variable decides *where they run*, it feeds `worldParameters.launchUrl`, which every relative-path step (`Given I am on "/path"`) resolves against. Keep them paired in your pipeline definitions: the job that sets `LAUNCH_URL` to production is the same job that passes `--tags "@production and @readonly"`, so the safe filter cannot be forgotten separately from the dangerous URL. The `@production and @readonly` double-tag is deliberate defense in depth, a scenario must be explicitly marked safe twice before it

touches the live site. And prefer relative paths in scenarios over the hardcoded URLs shown in the staging example above; a scenario that bakes in `staging.example.com` can never be reused as a production health check.

At scale: Environment wiring is where DevOps and QA meet, and the clean handshake is: QA owns the tags, DevOps owns the variables. CI injects `LAUNCH_URL`, credentials, and `BROWSER` per environment from pipeline secrets; the repository contains zero environment-specific URLs. Do this on all your projects and the production smoke check becomes a one-template cron job, the same `--tags "@production and @readonly"` line, twenty different `LAUNCH_URL` values, one wake-up call when any of them goes red.

Scaling the Suite

Recipe 183 Parallel Execution

Problem: The nightly regression takes fifty minutes, the team keeps adding scenarios, and the morning standup starts before the run finishes. Sequential execution means your feedback loop grows linearly with your coverage, the better your testing gets, the slower your answers arrive. Most of that time the CPU is idle, waiting on page loads. Parallel workers turn that waiting into throughput.

Solution:

Cucumber-js supports parallel execution:

```
# Run with 4 parallel workers
npx cucumber-js --parallel 4

# Run with workers matching CPU cores
npx cucumber-js --parallel $(nproc)
```

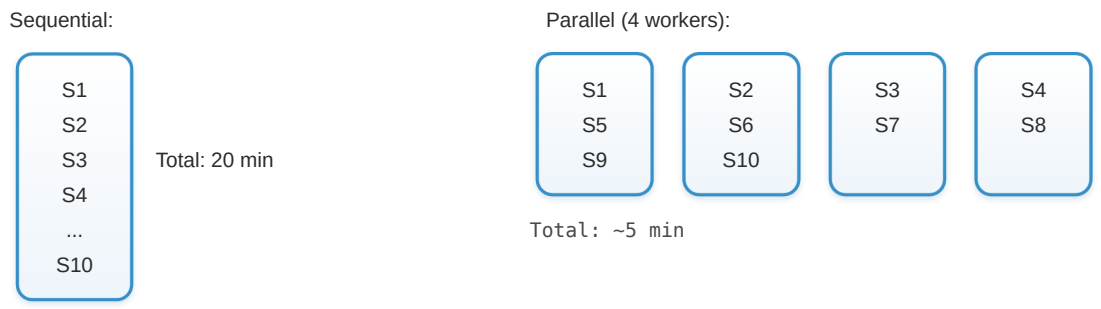


FIGURE 19.3 *Parallel Execution (Recipe 183)*

Requirements for parallel execution:

- Each scenario must be fully independent (no shared state)
- Database operations should not conflict between workers
- File paths should be unique per worker (for screenshots, reports)
- webship-js already ensures this: each scenario gets its own browser

How It Works: `--parallel N` spawns N worker processes, and Cucumber hands each worker the next pending scenario as it frees up, dynamic distribution, so one slow feature does not strand a worker while others sit idle. Each worker is a full process with its own browser, its own World, and its own `BeforeAll` / `AfterAll` (remember Recipe 176: "once" means once per worker). Browser isolation comes free from the webship-js lifecycle, which leaves *application state* as your real risk: two workers registering `test-user@example.com` at the same moment will collide in the database even though their browsers never met. Make test data unique per scenario, a timestamp or random suffix in usernames is usually enough. Speedup is sublinear: browsers are memory-hungry, so on a typical 4-core CI runner, 3-4 workers is the sweet spot before swapping eats your gains.

Pro Tip: When a scenario fails only under `--parallel`, do not tag it `@flaky` and move on, it is almost always a shared-state bug telling you two scenarios touch the same record. Fix the data collision and you have made the suite safer everywhere, not just faster.

Recipe 184 Sharding Tests Across CI Workers

Problem: You have maxed out `--parallel` on one runner, the machine is out of cores and memory before you are out of scenarios, and the release-candidate run still takes forty minutes. The next ceiling to break is the machine itself: split the feature files across four CI machines that run simultaneously, and the wall-clock time becomes whatever the slowest shard takes.

Solution:

```
# GitHub Actions matrix strategy
jobs:
  test:
    strategy:
      matrix:
        shard: [1, 2, 3, 4]
    steps:
      - name: Run shard ${{ matrix.shard }} of 4
        run: |
          # Get all feature files
          FEATURES=$(find tests/features -name "*.feature" | sort)
          TOTAL=$(echo "$FEATURES" | wc -l)
          PER_SHARD=$(( (TOTAL + 3) / 4 ))  # Ceiling division
          START=$(( (${{ matrix.shard }} - 1) * PER_SHARD + 1 ))
          SHARD_FEATURES=$(echo "$FEATURES" | sed -n "${START},$((START +
PER_SHARD - 1))p")
          npx cucumber-js $SHARD_FEATURES
```

How It Works: The matrix strategy launches four identical jobs that differ only in the `shard` variable. Each job lists all feature files, sorts them, the `sort` is essential, because the split is only deterministic if every machine sees the same order, and takes its quarter by line range. There is no coordinator and no shared state between machines, which is what makes this trivially portable to GitLab CI, Azure Pipelines, or any of the other CI configs webship-js ships. The trade-off versus `--parallel` is balance: sharding splits by *file count*, not by runtime, so a shard holding your three heaviest features finishes last while others idle. Two mitigations: combine both levels (`--parallel 2` inside each shard smooths the imbalance), and if one feature file dominates, split it

into smaller files, which usually improves organization anyway. Each shard writes its own `cucumber_report.json`, so add a final job that collects the four artifacts and merges them before report generation, or you will have four partial answers and no single verdict.

At scale: Sharding multiplies cost visibly: four machines for ten minutes is the same compute bill as one for forty, plus four browser-image pulls and four `npm install`s of overhead. Reserve full sharding for the suites and the moments that need it, release candidates, the two or three heaviest products, and let small projects run unsharded. And pin the browser image version across all shards in the template; a shard farm where machine 3 pulled a newer Chromium than the others produces the most confusing “flaky” failures you will ever debug.

Recipe 185 Running Specific Features

Problem: You are iterating on one failing scenario, and every check costs a full-suite run, twenty minutes to learn whether a one-line selector fix worked. That feedback loop trains people to push and pray instead of verifying locally. The fix is surgical targeting: run exactly the file, folder, scenario, or tag you are working on, and save the full run for CI.

Solution:

```
# Single feature file
npx cucumber-js tests/features/auth/login.feature

# All features in a directory
npx cucumber-js tests/features/auth/

# Multiple specific files
npx cucumber-js tests/features/auth/login.feature
tests/features/products/cart.feature

# By tags
npx cucumber-js --tags "@smoke"
npx cucumber-js --tags "@auth and not @wip"

# Specific scenario by line number
npx cucumber-js tests/features/auth/login.feature:15
```

How It Works: Path arguments *replace* the default feature glob from cucumber.js, while everything else in the profile, step definitions, the 45s timeout, the JSON report target, still loads, so a targeted run behaves identically to the full suite, just smaller. The `file.feature:15` form runs the single scenario whose declaration spans that line; failed-run output prints these `path:line` references precisely so you can paste one back and re-run only what broke. Combine dimensions freely: `npx cucumber-js tests/features/auth/ --tags "not @wip"` intersects a folder with a tag filter. The debugging trifecta on a single scenario is `HEADLESS=false npx cucumber-js tests/features/auth/login.feature:15`, a visible browser (with the headed-mode 800ms SLOW_MO making each action watchable) on exactly the failing case, seconds after CI told you the line number.

Chapter Summary

- Organize features by business domain (auth/, products/, admin/)
- Organize step definitions by domain, not by keyword
- Use a consistent tag naming convention across the team
- Implement test suite tiers: smoke (fast), regression (thorough), full (comprehensive)
- Environment tags prevent running dangerous tests in production
- Parallel execution with `--parallel N` speeds up long suites
- CI sharding distributes tests across machines for maximum speed
- Each scenario must be independent for parallel/shard execution to work
- Run specific files, folders, or tagged subsets as needed

Next: Design Patterns for Tests, proven patterns for maintainable test code.

Chapter 20: Design Patterns for Tests

Bad test code is worse than no test code. It gives a false sense of security while being so brittle and unmaintainable that the team eventually abandons it. Good test design means your tests survive refactors, communicate intent clearly, and run reliably.

This chapter covers the design patterns that make test suites maintainable at scale.

Pattern Decision Tree

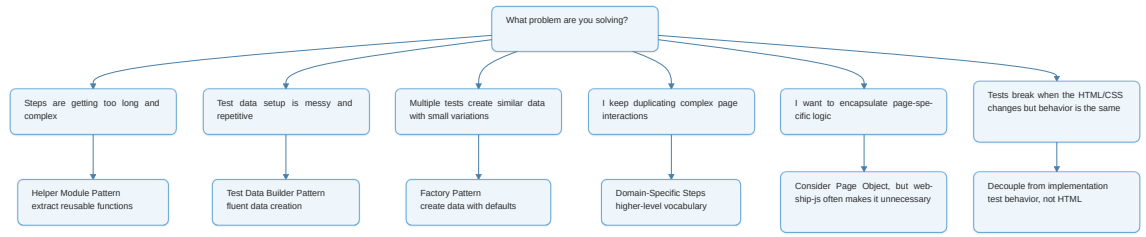


FIGURE 20.1 *Pattern Decision Tree*

Recipe 186 Helper Module Pattern

Problem: Your app's date picker takes four careful operations to drive, open, wait for the popup, click the date, wait for it to close. That sequence now lives copy-pasted in five step definitions, and four of the copies got the fix when someone added the close-wait; the fifth is your newest flaky test. Logic that exists in more than one place ages at different rates in each.

Solution:

```
// tests/support/helpers.js
module.exports = {
  async waitForTableToLoad(page) {
    await page.locator('table tbody tr').first().waitFor({ timeout: 10000
});
  },

  async getTableRowCount(page) {
    return await page.locator('table tbody tr').count();
  },

  async selectDateInPicker(page, dateString) {
    await page.locator('.date-picker-trigger').click();
    await page.locator('.date-picker-popup').waitFor({ state: 'visible'
});
    await page.locator(`[data-date="${dateString}"]`).click();
    await page.locator('.date-picker-popup').waitFor({ state: 'hidden' });
  },

  async dismissAllNotifications(page) {
    const notifications = page.locator('.notification .close');
    const count = await notifications.count();
    for (let i = 0; i < count; i++) {
      await notifications.nth(i).click();
    }
  },
};
```

```
// tests/step-definitions/admin-steps.js
const { When, Then } = require('@cucumber/cucumber');
const assert = require('assert');
const { waitForTableToLoad, getTableRowCount, selectDateInPicker } =
require('../support/helpers');

When('I filter by date {string}', async function (date) {
  await selectDateInPicker(this.page, date);
});

Then('the table should have {int} rows', async function (expected) {
  await waitForTableToLoad(this.page);
  const count = await getTableRowCount(this.page);
  assert.strictEqual(count, expected);
});
```

When to use helpers:

- Same logic appears in 3+ step definitions
- The logic is complex enough to benefit from a descriptive function name
- The function is reusable across different step definition files

How It Works: Helpers are plain functions that receive `page` as an argument instead of reading `this`, a deliberate choice. Step definitions need Cucumber's World binding; helpers that take explicit parameters can be called from any step file, any hook, even a one-off script, and they are unit-testable in isolation. Notice `selectDateInPicker` waits for the popup to be *hidden* before returning, it encodes the widget's full lifecycle so callers cannot race it, which is exactly the knowledge that was rotting in five copies. Keep the layering straight: Gherkin says *what* the business does, the step definition translates, the helper knows *how the widget works*. When a helper starts containing assertions or business decisions, it has climbed a layer too high.

At scale: The helpers file is your portfolio's pressure gauge. When `dismissAllNotifications` shows up in the third project, that is the signal to move it, and its friends, into a shared internal npm package versioned like any other dependency. One fix to the date-picker lifecycle then reaches twenty suites through a version bump instead of twenty pull requests. The discipline that makes this work: helpers take everything through parameters, never reach into project-specific config.

Recipe 187 Test Data Builder Pattern

Problem: A scenario about blocked users needs one thing to be true, the user is blocked, but the object literal that creates the user spells out seven fields, six of which are noise. Multiply that across forty scenarios and two problems compound: nobody can tell which field a test actually cares about, and the day the user model gains a required `department` field, all forty literals need editing by hand.

Solution:

```
// tests/support/builders.js

class UserBuilder {
  constructor() {
    this.data = {
      firstName: 'Test',
      lastName: 'User',
      email: `test-${Date.now()}@example.com`,
      role: 'viewer',
      status: 'active',
    };
  }

  withName(first, last) {
    this.data.firstName = first;
    this.data.lastName = last;
    return this;
  }

  withEmail(email) {
    this.data.email = email;
    return this;
  }

  asAdmin() {
    this.data.role = 'admin';
    return this;
  }

  asEditor() {
```

```
    this.data.role = 'editor';
    return this;
  }

  blocked() {
    this.data.status = 'blocked';
    return this;
  }

  build() {
    return { ...this.data };
  }
}

class ProductBuilder {
  constructor() {
    this.data = {
      name: 'Test Product',
      price: 9.99,
      sku: `SKU-${Date.now()}`,
      category: 'General',
      stock: 100,
    };
  }

  withName(name) { this.data.name = name; return this; }
  withPrice(price) { this.data.price = price; return this; }
  inCategory(cat) { this.data.category = cat; return this; }
  outOfStock() { this.data.stock = 0; return this; }
  build() { return { ...this.data }; }
}

module.exports = { UserBuilder, ProductBuilder };
```

```
// Usage in step definitions
const { Given } = require('@cucumber/cucumber');
const { UserBuilder } = require('../support/builders');
const axios = require('axios');

Given('there is an admin user named {string}', async function (name) {
  const [first, last] = name.split(' ');
  const user = new UserBuilder()
    .withName(first, last)
    .asAdmin()
    .build();

  await axios.post(this.launchUrl + '/api/test/create-user', user);
  this.createdUser = user;
});

Given('there is a blocked user', async function () {
  const user = new UserBuilder().blocked().build();
  await axios.post(this.launchUrl + '/api/test/create-user', user);
  this.createdUser = user;
});
```

Builder vs Object Literal:

```
Object Literal (fragile):            Builder (flexible):
const user = {                       const user = new UserBuilder()
  firstName: 'John',                   .withName('John', 'Doe')
  lastName: 'Doe',                     .asAdmin()
  email: 'john@e.com',                .build();
  role: 'admin',
  status: 'active',                  // Only specify what matters
  // Must specify ALL fields         // Defaults handle the rest
};                                   // Fluent, readable API
```

How It Works: Three details carry the pattern. Every setter ends with `return this`, which is all "fluent" means, each call hands back the builder so the next call can chain. The constructor owns the defaults, so a new required field is added in exactly one place and every existing test keeps passing. And `build()` returns `{ ...this.data }`, a copy, so a builder can be reused without two tests mutating the same object behind each other's backs. The `Date.now()` in the

default email is quietly the most important line: unique-by-default data is what makes scenarios safe to run under `--parallel` (Recipe 183) and immune to Monday's leftovers (Recipe 177). Note the usage steps stash the result in `this.createdUser`, the World carries it to later steps and to a tagged After hook that knows what to delete.

Pro Tip: Name builder methods in business language, `asAdmin()`, `blocked()`, not `setRole('admin')`, `setStatus('blocked')`. The chain then reads like the scenario it serves, and a reviewer can verify intent without checking what role string "blocked" maps to.

Recipe 188 Factory Pattern for Test Data

Problem: Builders are great, but for simple objects they are ceremony, a class, six methods, and a `build()` call to express "a user who is an admin". When a scenario needs three users and two products as supporting cast, you want test data the way you want coffee on a release morning: fast, with sensible defaults, no ritual.

Solution:

```
// tests/support/factories.js

let counter = 0;

const factories = {
  user(overrides = {}) {
    counter++;
    return {
      firstName: 'Test',
      lastName: `User${counter}`,
      email: `test-user-${counter}-${Date.now()}@example.com`,
      role: 'viewer',
      status: 'active',
      ...overrides,
    };
  },

  product(overrides = {}) {
    counter++;
```

```
    return {
      name: `Product ${counter}`,
      price: 19.99,
      sku: `SKU-${counter}-${Date.now()}`,
      category: 'General',
      stock: 100,
      ...overrides,
    };
  },

  order(overrides = {}) {
    counter++;
    return {
      orderId: `ORD-${counter}`,
      status: 'pending',
      total: 0,
      items: [],
      ...overrides,
    };
  },
};

module.exports = factories;
```

```
// Usage
const { user, product } = require('../support/factories');

// Default user
const viewer = user();
// { firstName: 'Test', lastName: 'User1', email: 'test-user-1-...', role:
'viewer' }

// Admin user
const admin = user({ role: 'admin', firstName: 'Rajab' });
// { firstName: 'Rajab', lastName: 'User2', email: 'test-user-2-...',
role: 'admin' }

// Expensive product
const laptop = product({ name: 'Laptop Pro', price: 999.99 });
```

How It Works: The whole pattern is one line of JavaScript: `{ ...defaults, ...overrides }`. Spread the defaults first, spread the overrides second, and later keys win, so `user({ role: 'admin' })` replaces only the role and inherits everything else. Uniqueness comes from two sources stacked together: the module-level `counter` guarantees no collisions within one worker process, and `Date.now()` in the email keeps two parallel workers (each with its own module instance, so each with its own counter starting at zero) from minting the same address. Choose by axis of variation: factories shine when tests differ in *field values* (override and go), builders earn their ceremony when tests differ in *kinds of setup* and named methods like `blocked()` document the business meaning. Most suites end up with factories for the supporting cast and builders for the protagonist.

At scale: Factories quietly define your team's "default user" and "default product". Standardize those defaults across projects, same shape, same `test-` email convention, and every engineer reads every suite's data setup at a glance, while the nightly cleanup job (Recipe 177) can sweep all projects with one marker query. Keep factories *thin*: they return plain objects and never call the API themselves, so the same factory feeds an API-seeding step on one project and a form-filling step on another.

Recipe 189 The Page Object Pattern (When to Use with webship-js)

Problem: A new teammate arrives from a Selenium shop and their first PR wraps every page in a Page Object class, including the login form that webship-js's built-in steps already handle in three Gherkin lines. Now there are two ways to fill a field in the same suite, and every future test author has to choose. The question is not whether Page Objects are good; it is where they still earn their keep in a stack that already abstracts selectors twice.

Solution:

The Page Object pattern is common in Selenium-based testing. In webship-js, it is **less necessary** because:

1. webship-js steps already abstract selectors (you write "fill in 'Email'" not `page.locator('#email')`)
2. Gherkin already provides a domain-language layer
3. Playwright's `getByLabel`, `getByRole`, `getByText` are resilient to HTML changes

However, for complex pages with custom widgets, Page Objects can help:

```
// tests/support/pages/checkout-page.js
class CheckoutPage {
  constructor(page) {
    this.page = page;
  }

  async fillShippingAddress(address) {
    await this.page.getByLabel('Street').fill(address.street);
    await this.page.getByLabel('City').fill(address.city);
    await this.page.getByLabel('State').fill(address.state);
    await this.page.getByLabel('ZIP').fill(address.zip);
  }

  async selectShippingMethod(method) {
    await this.page.getByLabel(method).check();
  }

  async getOrderTotal() {
    return await this.page.locator('.order-total').textContent();
  }

  async placeOrder() {
    await this.page.getByRole('button', { name: 'Place Order' }).click();
    await this.page.waitForLoadState('networkidle');
  }
}

module.exports = CheckoutPage;
```

```
// tests/step-definitions/checkout-steps.js
const { When, Then } = require('@cucumber/cucumber');
const CheckoutPage = require('../support/pages/checkout-page');

When('I complete the checkout with standard shipping', async function () {
  const checkout = new CheckoutPage(this.page);
  await checkout.fillShippingAddress({
    street: '123 Test St',
    city: 'Austin',
    state: 'TX',
    zip: '78701',
  });
  await checkout.selectShippingMethod('Standard Shipping');
  await checkout.placeOrder();
});
```

When Page Objects help:

USE Page Objects when:

- Complex custom widgets (date pickers, rich editors)
- Page has many selectors that change frequently
- Multiple step definitions interact with the same page
- Team has Selenium experience

SKIP Page Objects when:

- Standard forms (use webship-js built-in steps)
- Simple pages
- webship-js steps already cover the interaction
- You are adding complexity without benefit

FIGURE 20.2 *The Page Object Pattern (When to Use with webship-js) (Recipe 189)*

How It Works: Page Objects were invented to solve a Selenium-era problem: raw `driver.findElement` calls scattered through tests, so selectors needed a home. webship-js already gives selectors two homes. The built-in steps resolve fields label-first (`When I fill in "Email"` finds the input by its label, then falls back to id, name, placeholder), and the named-selectors registry maps stable names to selectors in JSON files under `tests/selectors/`, auto-

loaded, referenced by name in any step, changeable without touching code. That covers most of what a Page Object class used to do, with less machinery. What remains is genuinely multi-step *choreography*: a checkout page where address, shipping method, and order placement always travel together. The `CheckoutPage` class bundles that choreography behind methods, and the step definition stays one readable paragraph. The honest test for whether you need one: if the class would only rename Playwright calls one-to-one, it is indirection, not abstraction, skip it.

At scale: Whatever you decide, decide it once and write it down. The expensive outcome across twenty projects is not “Page Objects” or “no Page Objects”, it is ten projects that wrap everything and ten that wrap nothing, so no engineer’s instincts transfer between them. A two-paragraph convention (“built-in steps and named selectors by default; a page class only for widgets with multi-step lifecycles”) in your shared testing guide settles every future PR debate before it starts.

Recipe 190 The Fixture Antipattern (and What to Do Instead)

Problem: Someone seeds a `fixtures.sql` with “the standard test data” and for a month life is good. Then test A deletes user 2, test B still expects user 2, and the suite starts failing only when both run, in that order. Soon there is an unwritten law that scenarios must run alphabetically, a `users` table nobody dares add a column to, and a Slack thread titled “why does checkout.feature fail on Thursdays?”. The fixture became a load-bearing wall everyone is afraid to touch.

Why it is dangerous:

The Fixture Trap:

```
Shared fixtures.sql:
INSERT INTO users (id, name) VALUES (1, 'Admin');
INSERT INTO users (id, name) VALUES (2, 'Editor');
INSERT INTO products (id, name) VALUES (1, 'Laptop');

Problem 1: Test A deletes user 2
Problem 2: Test B expects user 2 to exist
Result: Tests pass individually, fail together

Problem 3: Someone adds a field to the users table
Problem 4: Fixture SQL does not include the new field
Result: ALL tests using the fixture break

Problem 5: Tests depend on specific IDs (user 1, product 1)
Problem 6: Adding a new fixture row shifts IDs
Result: Cascading failures across the suite
```

FIGURE 20.3 *The Fixture Antipattern (and What to Do Instead) (Recipe 190)*

What to do instead:

1. **Each test creates its own data** (via API, UI, or Builder pattern)
2. **Each test cleans up its own data** (via After hook or API call)
3. **Tests never assume existing data** (they create what they need)

```
# BAD: Assumes fixture data exists
Scenario: Edit user
  When I go to "/admin/users"
  And I click "Edit" in the "admin@example.com" row
  # What if fixture does not include this user?

# GOOD: Creates its own data
Scenario: Edit user
  Given I am a logged in user with the "admin" user
  And there is a user named "John Smith" with email "john@test.com"
  When I go to "/admin/users"
  And I click "Edit" in the "john@test.com" row
  # Test created its own data, guaranteed to exist
```

How It Works: The root failure is hidden coupling: the fixture is shared mutable state, and every test that reads or writes it is invisibly wired to every other. Self-created data cuts the wires, each scenario builds exactly what it asserts on (a factory plus one API call, milliseconds of overhead), so scenarios become order-independent, which is precisely the property `--parallel` and CI sharding require. This is also the only model where a failure means something: when the "Edit user" scenario goes red, the user *was* created two lines earlier, so the bug is in editing, not in some fixture drift three weeks old. One pragmatic boundary: reference data the application ships with (countries, roles, default config) is fine to assume, because the app owns it and migrates it. The rule is narrower and sharper, never assume *records that some test or human put there*.

At scale: Shared staging environments make the fixture trap deadlier: now it is not two tests coupled through one database, it is two *projects*. Self-created, uniquely-named data (the `Date.now()` emails from Recipes 187-188) is the only pattern that survives twenty suites pointing at overlapping environments. And when DevOps rebuilds staging from scratch, fixture-dependent suites all break at once while self-seeding suites pass on the first run, that one morning pays for the whole discipline.

Recipe 191 DAMP Over DRY in Tests

Problem: A scenario fails in the nightly run and the engineer on triage duty opens the feature file to find: `Given the standard test setup`, `And the admin is ready`, `When I do the thing`. Three files later they finally know what the test was checking, at 8 a.m., the Tuesday before release. The suite was refactored by someone who treated Gherkin like production code and extracted every repeated line, and in the process deleted the one thing a test must keep: the ability to be understood on its own.

The DAMP principle: Descriptive And Meaningful Phrases. In test code, clarity is more important than eliminating duplication.

```
# TOO DRY (hard to understand in isolation):
Background:
  Given the standard test setup
  And the admin is ready

Scenario: Test something
  When I do the thing
  Then it should work

# DAMP (clear, self-contained):
Scenario: Admin can create a product
  Given I am a logged in user with the "admin" user
  When I go to "/admin/products"
  And I press "Add Product"
  And I fill in "Name" with "Wireless Mouse"
  And I fill in "Price" with "29.99"
  And I press "Save"
  And wait for AJAX to finish
  Then I should see "Product created"
```

DRY (DON'T REPEAT YOURSELF)	DAMP (DESCRIPTIVE AND MEANINGFUL)
Minimize duplication	Maximize readability
Abstract shared steps	Repeat for clarity
Short, referencing	Self-contained, standalone
Good for production code	Good for test code
Bad for test readability	Tests are documentation

A test should be understandable by reading ONLY that test. If you need to look at the Background, a helper, and three other files to understand what a test does, it is too DRY.

How It Works: Production code and test code optimize for different failure modes. Production code is *executed* constantly and *read* occasionally, so duplication is the enemy, a bug fixed in one copy survives in the others. Test code is the mirror image: it is *read* at the worst moments (failures, audits, “do we cover this?” questions) by people who did not write it, so opacity is the enemy. The resolution is to put the DRY in the right layer. `Given I am a logged in user with the "admin" user` repeated across thirty scenarios is not duplication, it is one step definition referenced thirty times, and renaming the login button is still a one-line fix. The Gherkin stays DAMP, the JavaScript underneath stays DRY, and both principles get what they want. The honest test: hand a failing scenario to someone who has never seen the repo, alone, and ask them what the application failed to do. If they can answer, the altitude is right.

At scale: DAMP scenarios are what make your suite legible to people who will never read JavaScript, the product owner signing off on acceptance criteria, the delivery manager reading the HTML report after a red nightly, the auditor sampling your coverage. A scenario that says `Given the standard test setup` is engineering shorthand; a scenario that says exactly who logs in and what they do is *evidence*. Across a portfolio, that difference decides whether your test reports are something managers can read or something they have to take on trust, and “a green run they can read themselves” is the entire point of writing these tests in Gherkin.

Chapter Summary

- Use the Pattern Decision Tree to choose the right pattern for your problem
- Helper Modules extract reusable functions called by step definitions
- Test Data Builders provide fluent APIs for complex test data
- Factories create data with sensible defaults and optional overrides
- Page Objects are less necessary with webship-js but useful for complex widgets
- Avoid shared fixtures, each test should create and clean up its own data
- DAMP (Descriptive And Meaningful Phrases) beats DRY in test code
- Tests are documentation, optimize for readability, not code reuse

This concludes Part IV: Architecture and Organization.

Next: When Tests Go Bad, diagnosing and fixing flickering, brittle, and slow tests.

Part V: Troubleshooting and Maintenance

Chapter 21: When Tests Go Bad

Your test suite was green. Every scenario passed. The team trusted it. Then, slowly, things started to rot. A test fails on Monday but passes on Tuesday. A tiny UI change breaks twenty scenarios. The suite takes forty-five minutes to run. Stakeholders stop reading the feature files. The team stops fixing failures, marking them as `@flaky` and moving on.

This is the natural lifecycle of an unmaintained test suite, and on a multi-project team it never happens to just one suite. The habits that let one suite rot (rerun it, tag it, mute it) travel to the next project along with the copied step definitions, until the team spends its mornings triaging twenty red pipelines instead of testing anything. This chapter is the antidote. We will diagnose each symptom, identify the root cause, and apply the fix, and for each fix, note what changes when you apply it across a portfolio instead of a single project.

The Four Symptoms

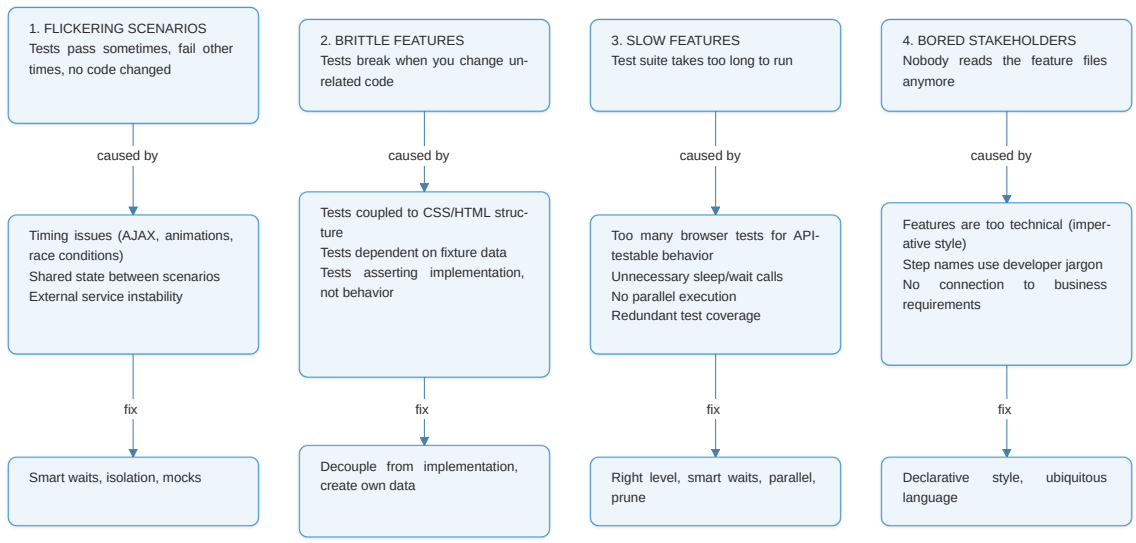


FIGURE 21.1 *The Four Symptoms*

Flickering Scenarios

Recipe 192 Diagnosing Timing Issues

Problem: It is the Tuesday before release. The login scenario that has been green for three months fails in CI, passes on retry, then fails again on a different project. Nobody changed the test. Your QA lead has twenty suites to watch and no time to babysit one, so the team starts muttering “just rerun it”, which is how trust in the whole suite dies. Before you touch any code, you need to know which kind of timing problem you are looking at.

Diagnosis checklist:

```
Is the failure related to:
  □ Content not found ("should see X" fails)?
    → The update arrived through a channel smartSettle cannot count
      (WebSocket push, setInterval polling) -- assert with a web-first
      step that polls, instead of a one-shot check
  □ Element not clickable?
    → Element is behind an overlay, or animation not complete
  □ Stale element reference?
    → DOM changed between locating and interacting
  □ Different on CI than locally?
    → CI is slower; the work finished after the settle budget --
      extend the assertion window with "within N seconds"
```

Solution:

```
# BAD: one-shot assertion the instant the click returns
When I press "Save"
Then I should see "Saved"

# GOOD: web-first assertion -- polls every 100ms for up to 5 seconds
When I press "Save"
Then ".messages" should contain text "Saved"

# GOOD: when the server is genuinely slow, widen the window
When I press "Save"
Then ".messages" should contain text "Saved" within 15 seconds
```

How It Works: webship-js 2.0 has two layers of defense against timing flake, and diagnosing means figuring out which layer the failure escaped. The first layer is `smartSettle`. On every page load, an init script installs three counters in the browser: `window.__webshipAjaxCount` (wraps fetch and XHR), `window.__webshipPendingTimers` (wraps setTimeout), and `window.__webshipLastMutation` (a MutationObserver timestamp). After every state-changing step, an AfterStep hook runs `smartSettle(page, 1500)`: it polls those counters every 50ms and considers the page settled only when AJAX count is zero, pending timers are zero, AND the DOM has been quiet for at least 250ms. Navigations get a 3000ms budget. The second layer is

web-first assertions: steps like `should contain text` re-check every 100ms for up to 5 seconds (or the `within N seconds` you specify). A test that flakes despite both layers usually means the update came through a channel the counters cannot see, a WebSocket push, a long setInterval, a server-sent event, or the work simply took longer than the budgets on a slow CI runner. The fix is never a sleep; it is a web-first assertion with a window wide enough for your slowest environment.

At scale: When you run twenty projects through the same CI farm, log every flaky failure with the project name and the failing step before you fix anything, the `--metadata Key=Value` flags on `bin/generate-reports.js` let you stamp each report with project and environment. Within a month you will see the pattern: it is almost never twenty different timing bugs, it is two or three (one slow staging server, one WebSocket-heavy app) showing up everywhere. Fix the category, not the scenario.

Recipe 193 Fixing Race Conditions

Problem: The checkout test clicks “Place Order” and immediately asserts “Order Confirmed”. On your laptop the server answers in 80ms and the test passes every time. On the shared CI runner, the one all your projects queue on at 2 a.m., the server takes 1.8 seconds, the assertion fires first, and the scenario fails. This is the classic race: the test and the application are running at different speeds, and the test assumed it would win.

Solution: older suites patched this with the “double-wait pattern”, wait for AJAX, then sleep one more second for the DOM. webship-js 2.0 bans static sleeps outright, because the framework already closes both gaps:

```
# The 2.0 way: act, then assert with a web-first step
When I press "Place Order"
Then ".order-status" should contain text "Order Confirmed"
```

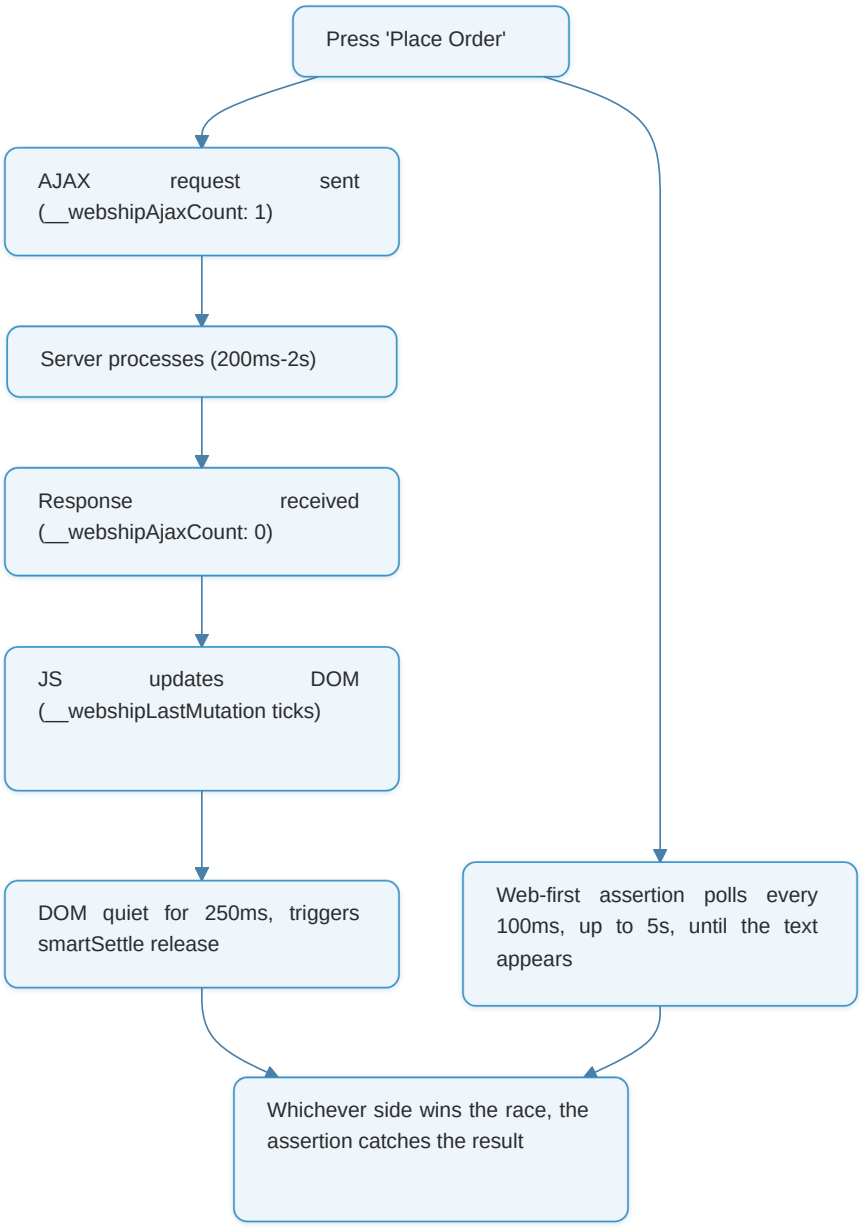


FIGURE 21.2 *Fixing Race Conditions (Recipe 193)*

How It Works: the old "wait 1s" was a guess, it covered the DOM-update gap on your machine and nobody else's. smartSettle removes the guess: the AfterStep hook after `I press` polls `window.__webshipAjaxCount`, `window.__webshipPendingTimers`, and `window.__webshipLastMutation` every 50ms and only releases the next step when requests are

drained, timers are empty, and the DOM has been quiet for 250ms (budget 1500ms per step). The web-first assertion then covers whatever is left: it re-evaluates the locator every 100ms for up to 5 seconds. Between the two, there is no moment where "the response arrived but the DOM has not painted yet" can fail your test. If you ever feel tempted to add a sleep, that is the signal you are asserting on the wrong thing, assert on the outcome the user sees, not on the moment you expect it.

At scale: race conditions are the number-one source of "passes locally, fails in CI" tickets across a multi-project portfolio, because every project's staging server has a different latency profile. Make "no static sleeps, assert the outcome" a written rule in your shared step-definition conventions and enforce it in PR review, one team did this across the Varbase distribution's test suites and the cross-site flake rate dropped before anyone tuned a single timeout.

Recipe 194 Eliminating Shared State

Problem: Every scenario passes when you run it alone. Run the full suite and three of them fail, but only in a specific order, and only sometimes, because `--parallel` shuffles the order on every run. The new hire who wrote the third scenario gets blamed, even though the real culprit is a scenario written eight months ago that leaves a product behind in the database. Order-dependent failures are the hardest flake to spot because the failing test is innocent.

Diagnosis:

```
# Run the failing test alone
npx cucumber-js tests/features/specific.feature

# Run with the suspected polluter
npx cucumber-js tests/features/polluter.feature
tests/features/specific.feature

# If it fails in the second case, polluter is creating shared state
```

Solution:

- Each scenario creates its own data
- After hooks clean up created data
- Each scenario gets a fresh browser (webship-js does this by default)
- Never write to global variables in step definitions

How It Works: webship-js isolates everything it controls: each scenario gets a fresh browser context, so cookies, localStorage, and sessions never leak between scenarios, and `Given I am an anonymous user` explicitly clears cookies before navigating to the launch URL. What the framework cannot isolate is *your application's database*. If scenario A creates "Test Product" and never deletes it, scenario B's assertion `should have a count of 1` finds two. The bisect technique above works because shared state needs both a writer and a reader: run them together and the failure reproduces deterministically, which turns a flaky mystery into an ordinary bug. Also watch for module-level variables in step definition files, step files are loaded once per worker, so a `let lastOrderId` at the top of a file silently couples every scenario that worker runs.

At scale: shared state is what kills parallel execution, and parallel execution is what keeps forty nightly suites affordable. Make data isolation a gate, not a guideline: every project's CI should run the suite with `--parallel` from day one, so order-dependence surfaces in week one of a project instead of month six. A scenario that only passes sequentially is a defect, even while it is green.

Recipe 195 Handling External Service Flakiness

Problem: Your checkout suite went red at 3 a.m., not because checkout broke, but because the payment sandbox was down for maintenance. The next morning someone spends an hour confirming that nothing is wrong with your code. Multiply that across every project that touches a payment gateway, a map provider, or an email service, and you are burning a developer-day a week investigating other people's outages.

Solution:

```
Strategy 1: Mock the external service in test environment
Strategy 2: Add retry logic for known-flaky external calls
Strategy 3: Tag external-dependent tests and run separately
```

For the default run, mock the gateway at the network layer with the built-in routing steps, the test verifies *your* checkout flow, not their uptime:

```
Scenario: Checkout succeeds when the gateway approves the payment
  Given the URL "**/api/payments" returns the JSON:
    """
    { "status": "approved", "transactionId": "TX-1001" }
    """
  Given I am a logged in user with the "customer" user
  When I go to "/checkout"
  And I fill in "Card Number" with "4111111111111111"
  And I press "Pay"
  Then I should see "Payment Successful"
```

Keep one real-call scenario, tagged so it runs on its own schedule and its failures are triaged separately:

```
@external @flaky-ok
Scenario: Payment gateway processes credit card (real integration)
  # Runs nightly, not on every PR. A failure here means
  # "check the gateway status page" before "check our code".
  Given I am a logged in user with the "customer" user
  When I go to "/checkout"
  And I fill in "Card Number" with "4111111111111111"
  And I press "Pay"
  Then I should see "Payment Successful"
```

How It Works: `Given the URL "**/api/payments" returns the JSON:` registers a `page.route()` interceptor with a glob pattern, any request the browser makes that matches `**/api/payments` is answered from the docstring without ever leaving the machine. The same

step family can return error statuses (`returns status 500 with body`), block the URL entirely, or delay it by N milliseconds, which means you can also test how checkout behaves when the gateway is down, something you could never test reliably against the real sandbox. The mock removes the external dependency from the critical path; the tagged real-call scenario keeps one honest integration check so contract drift still gets caught.

At scale: agree on tag semantics once, across all projects: `@external` means “talks to a third party”, and CI runs `--tags "not @external"` on every PR while a nightly job runs the externals alone. Your delivery managers get a clean separation in the reports, “our code” failures versus “their service” failures, which is exactly the distinction they need when someone asks whether a release is safe.

Brittle Features

Recipe 196 Removing Fixture Data Dependencies

Problem: The suite was written against a hand-crafted staging database where user #1 was "John Smith". Then the third project you onboard gets a fresh database, user #1 is somebody else, and forty scenarios fail on day one, not because the application is broken, but because the tests memorized a database instead of describing behavior. Fixture-dependent tests only run in the one environment they were written against, which defeats the point of having them.

```
# BAD: Assumes user with id=1 exists
Scenario: View user profile
  When I go to "/users/1"
  Then I should see "John Smith"

# GOOD: Creates its own data
Scenario: View user profile
  Given there is a user named "John Smith"
  When I go to "/admin/users"
  And I click "View" in the "John Smith" row
  Then I should see "John Smith"
```

How It Works: the second scenario carries its own world with it. `Given there is a user named "John Smith"` creates the user, ideally through the API steps (`When I send a POST request to "/api/users" with values:`), which is faster and more stable than clicking through an admin form, and the rest of the scenario only ever references data it just created. That makes the scenario portable: it passes against a fresh database, a copy of production, a teammate's local DDEV site, and CI, without anyone preparing anything. The navigation step also matters: going through `/admin/users` and clicking the row exercises the real lookup path instead of hard-coding an ID that the test has no business knowing.

At scale: environment-independence is the single property that lets one team run twenty projects. The moment a suite needs "the staging database, in the right state", it can only run in one place, and your platform engineers cannot move it, parallelize it, or spin up review environments for it. Make "creates its own data" a checklist item in every test PR review (Recipe 215), it is much cheaper to enforce on the way in than to retrofit.

Recipe 197 Making Assertions Resilient

Problem: Marketing rewrites the cart message from "There are 5 items in your shopping cart" to "Your cart: 5 items". Nothing about the behavior changed, the count is right, the cart works, but a copy edit just broke twelve scenarios across three projects, and a content editor is now afraid to touch microcopy because "it breaks the tests". When wording changes cause failures, the tests are asserting prose, not behavior.

```
# FRAGILE: Exact text match
Then I should see "There are 5 items in your shopping cart"

# RESILIENT: Key information only
Then I should see "5 items"
Then I should see "shopping cart"
```

How It Works: `I should see` is a substring match against the page, so assert the smallest fragment that proves the behavior: the number "5 items" proves the count logic, "shopping cart" proves you are on the right surface. Everything else in the sentence is copywriting, and copywriting belongs to the content team, not the test suite. When you need more precision than a page-wide substring, scope it with a web-first step instead: `Then ".cart-count" should contain text "5"` polls that one element for up to 5 seconds and ignores the rest of the page. The judgment call: if a stakeholder would say "the feature still works", a wording change should not fail the test.

At scale: this matters double on multilingual and multi-brand portfolios. A suite that asserts full English sentences cannot run against the Arabic or French build of the same product. Assert numbers, codes, and element states where you can, and keep the few full-text assertions in a place translators know about.

Recipe 198 Decoupling from CSS/HTML Structure

Problem: The front-end team migrates from Bootstrap to Tailwind. Not one user-facing behavior changes, every button still submits, every form still validates, yet 200 scenarios fail because `.btn-primary` no longer exists. A refactor that should have been invisible to users was also supposed to be invisible to the tests. Tests coupled to CSS classes and DOM positions fail for reasons users would never notice.

```
// BAD: Coupled to CSS class
await page.locator('.btn-primary.submit-form.large').click();

// GOOD: Semantic selector
await page.getByRole('button', { name: 'Submit' }).click();

// BAD: Coupled to DOM structure
await page.locator('div.container > form > div:nth-child(3) >
input').fill('text');

// GOOD: Accessible selector
await page.getByLabel('Email').fill('text');
```

How It Works: webship-js already builds this resilience into its step definitions. `When I fill in "Email" with "..."` resolves the field label-first, it looks for a `<label>` with that text, then falls back to id, name, and placeholder, so the step survives any markup change that keeps the label intact. `When I click the "Submit" button` uses Playwright's `getByRole`, which reads the accessibility tree the same way a screen reader does: role plus accessible name. Both selectors only break when something a *user* would notice changes, the label text or the button name, which is exactly the property you want. As a bonus, suites written this way quietly enforce accessible markup: a button that `getByRole` cannot find is usually a button assistive technology cannot find either.

At scale: for the cases where you genuinely need a CSS or XPath selector (third-party widgets, canvas wrappers, legacy admin themes), do not scatter raw selectors through twenty projects' feature files. Put them in named-selector JSON files under `tests/selectors/` (`{"css": {...},`

`"xpath": {...}}`, auto-loaded via `worldParameters.selectors.files`) and reference them by name. When the markup changes, you edit one JSON entry per project, or one shared file, instead of grepping thirty repositories for `.btn-primary`.

Slow Features

Recipe 199 Replacing Sleeps with Smart Waits

Problem: You inherit a suite from a contractor and find `wait 5s` sprinkled through eighty scenarios, each one a fossil of some flake that someone "fixed" by waiting longer. The suite now takes forty minutes, most of it spent staring at pages that finished loading four seconds ago. Worse, the sleeps still flake on the slowest CI runner, because no fixed number is both fast and safe.

```
# BAD: Arbitrary sleep (always waits 5 seconds, even if done in 0.5)
And wait 5s
Then I should see "Result"

# GOOD: Assert the outcome -- web-first polling waits exactly as
# long as needed, and not a millisecond longer
Then ".results" should contain text "Result"
```

How It Works: webship-js 2.0 has a hard rule, no static sleeps anywhere, and the machinery to make it stick. Every state-changing step is followed by `smartSettle(page, 1500)`: it watches `window.__webshipAjaxCount`, `window.__webshipPendingTimers`, and the MutationObserver timestamp, and releases the moment requests are drained, timers are empty, and the DOM has been quiet for 250ms. Navigations get a longer ladder under a 3000ms budget, wait for the body to attach, then `domcontentloaded`, then a `networkidle` probe, then the same 50ms counter polling, and every gate exits the instant it is satisfied. If the page settles in 300ms, you wait 300ms, not 5 seconds. Web-first assertions cover the rest by polling every 100ms for up

to 5 seconds. So deleting a `wait 5s` is not a trade of speed against stability; the event-based waits are *both* faster on the happy path and more reliable on the slow one, because they key off what actually happened instead of a guess about how long it takes.

Impact calculation:

```
10 unnecessary "wait 5s" calls × 5 seconds each = 50 seconds wasted per
run
50 seconds × 100 runs per day = 83 minutes per day
83 minutes × 250 work days = 347 hours per year
```

At scale: now multiply that table by your project count. Forty suites with the same sleep habit waste hundreds of CI-hours a year, real money on metered runners, and real waiting on self-hosted ones where suites queue behind each other. A one-day "delete the sleeps" sweep across the portfolio, enforced afterward by a grep in PR review, is one of the highest-return maintenance tasks a QA lead can schedule.

Recipe 200 Reducing Browser Tests with API Tests

Problem: Your suite has eighty browser scenarios and most of them never needed a browser. Validation rules, CRUD round-trips, permission checks, all driven through page loads, form fills, and renders, at twenty seconds apiece. The suite outgrew the PR pipeline months ago, so it got demoted to nightly, which means developers now learn about breakage the morning after they caused it.

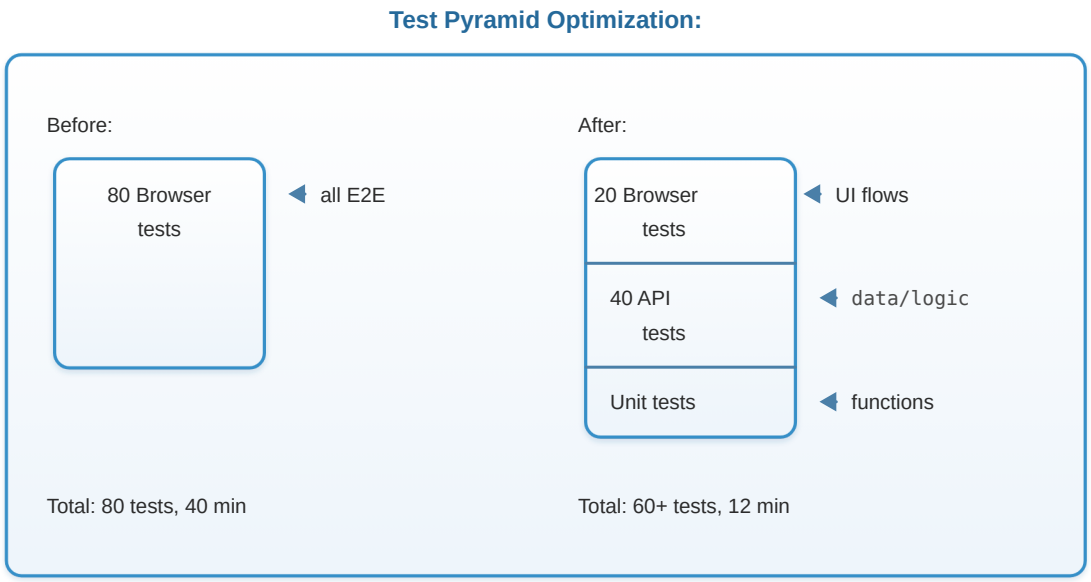


FIGURE 21.3 *Reducing Browser Tests with API Tests (Recipe 200)*

If a behavior can be tested via API (CRUD operations, validation, data transformations), test it via API. Reserve browser tests for:

- User workflows (login → navigate → interact → verify)
- Visual behavior (modals, animations, responsive layout)
- Interactions that require a real browser (file upload, drag-and-drop)

How It Works: the API steps stay in the same Gherkin suite, `Given the API base URL is "..."`, `When I send a POST request to "/api/users" with values:`, `Then the API response code should be 422`, so moving a test down the pyramid does not mean moving it to a different tool, a different report, or a different team. Under the hood the API steps go through axios, no browser launch, no rendering, no settle cycle: a validation check that took 20 seconds through the UI takes half a second as a request/response pair. The scenarios stay readable to the same product owners; they just stop paying the browser tax for behavior that never needed pixels.

At scale: the pyramid is also a triage policy. When an API scenario fails, the contract broke; when only the browser scenario fails, the UI broke. Reports that separate the two layers let a reviewer on another team localize a failure in seconds. And the runtime math compounds: moving half of each

suite's scenarios to the API layer across forty projects often buys back more CI time than any parallelism budget you could approve.

Recipe 201 Parallel Execution

Problem: Each suite is fine alone, but the nightly run executes them back to back and finishes at 7:40 a.m., after the team has started picking up tickets. The CI runners spend the night at 12% CPU, one scenario at a time, while everyone pays for the idle.

```
# Sequential: 20 minutes
npx cucumber-js

# 4 parallel workers: ~5 minutes
npx cucumber-js --parallel 4
```

How It Works: Cucumber's `--parallel N` starts N worker processes and deals scenarios out to them; each webship-js worker launches its own browser, so scenarios never share cookies, sessions, or pages. The speedup is close to linear until you saturate CPU or your application server, browsers are memory-hungry, so on a typical CI runner 4-6 workers is the sweet spot before you trade flake for speed. The one precondition is Recipe 194: scenarios must not share database state, because parallel order is effectively random. If turning on `--parallel` breaks scenarios, you have found real isolation bugs, not a parallelism problem.

At scale: parallelize within suites first, then across them, run multiple projects' suites simultaneously on separate runners or a CI matrix. Watch your application server while you tune: twenty workers hammering one shared staging site will produce timeouts that look like test flake but are really capacity. Your platform team would rather hear "we need a second staging instance" than chase phantom failures for a month.

Bored Stakeholders

Recipe 202 Writing Features Stakeholders Want to Read

Problem: Your product owner wrote the acceptance criteria for the catalog feature. Six weeks later she asks whether they are covered, and the QA engineer opens a feature file full of `#product_name` and "wait for the HTTP response". She cannot read it, so she cannot answer her own question, and from that day, sign-off happens on trust instead of on a green run she can verify herself. The whole point of Gherkin was to avoid exactly this.

```
# DEVELOPER PERSPECTIVE (boring to stakeholders):
Scenario: Fill form and POST
  Given I navigate to /admin/products/new
  When I input "Widget" into #product_name
  And I input "9.99" into #product_price
  And I submit the form
  And I wait for the HTTP response
  Then the page should contain "created"

# BUSINESS PERSPECTIVE (stakeholders care about this):
Scenario: Store manager adds a new product to the catalog
  Given I am logged in as a store manager
  When I add a new product "Widget" priced at $9.99
  Then the product should appear in the catalog
  And the inventory count should be zero (awaiting stock)
```

How It Works: the two scenarios run the same clicks; the difference is altitude. The first narrates the *mechanism*, selectors, HTTP, form internals, which means it must change whenever the mechanism changes and it tells a business reader nothing. The second narrates the *behavior*: who does what, and what the business outcome is. The mechanism moves into custom step definitions, where it belongs, and the feature file becomes a document a store manager can confirm or correct.

There is a discipline payoff too: declarative scenarios are naturally “one behaviour per scenario”, which keeps failure messages meaningful, “store manager cannot add a product” is a report headline; “could not find #product_price” is a debugging session.

At scale: readable features are how a PMO compares quality across projects without learning each codebase. When every project’s report reads as business behavior, a delivery manager can scan twenty HTML reports and answer “is release X safe?” from the scenario names alone. That only works if the writing standard is shared, which is the next recipe.

Recipe 203 Using Ubiquitous Language

Problem: The business calls them “customers”. The database calls them “users”. Half the feature files say one, half say the other, and in the sprint review a stakeholder asks, honestly, whether “user can place an order” and “customer can place an order” are two different features. Every vocabulary fork costs you a stakeholder’s confidence and a future maintainer’s afternoon.

Use the same terms the business uses:

- If they say “customer,” do not write “user”
- If they say “order,” do not write “transaction”
- If they say “publish,” do not write “set status to active”

How It Works: this is the “ubiquitous language” idea from domain-driven design applied to Gherkin: one vocabulary shared by the business, the code, and the tests. It pays off mechanically as well as socially. Step definitions match on text, so synonyms breed duplicate steps, `Given there is a customer ...` and `Given there is a user ...` become two implementations of the same thing, drifting apart over time. One term means one step, fewer definitions, and feature files a product owner can review for *correctness*, not just style. The cheapest way to start: lift the nouns and verbs directly from the acceptance criteria your product owners already write.

At scale: keep a short glossary per product, a page, not a wiki tree, and check feature files against it in review. On a multi-project team this is also an onboarding accelerator: a QA engineer rotating onto their third project already speaks the pattern, and only needs to learn the nouns.

Chapter Summary

- Four symptoms: flickering, brittle, slow, and boring tests
- Flickering: trust smartSettle plus web-first assertions, eliminate shared state, mock external services with the network routing steps
- Brittle: decouple from implementation, create test data per scenario, assert behavior not structure
- Slow: delete static sleeps, move tests to API level, run in parallel
- Bored stakeholders: write in business language, use declarative style, connect to requirements
- Every symptom has a diagnostic checklist and proven fixes
- webship-js already handles many common causes (per-scenario browser isolation, label-first and role-based selectors, automatic smartSettle waits)

Next: Debugging Techniques, tools and methods for finding why tests fail.

Chapter 22: Debugging Techniques

A test fails. The error message says `Expected to find "Dashboard" but it was not present`. That tells you what went wrong but not why. Was the page still loading? Did the login fail? Was there a server error? Is the selector wrong?

This chapter arms you with every debugging technique available in webship-js, Playwright, and Cucumber-js. One theme runs through all of them: on a team that owns many suites, the person debugging a failure is usually not the person who wrote the scenario. So every technique here is also a way of attaching evidence to a failure, screenshots, video, console errors, network logs, until a red build on a project you have never opened arrives with everything you need to triage it.

Recipe 204 Reading Cucumber Error Output

Problem: A red build lands in the team channel at 9 a.m. The reviewer on duty, who owns fifteen other suites and did not write this one, has thirty seconds to decide: real regression, environment hiccup, or broken test? The error text is the only evidence on the table, and most people skim past the one line that classifies the failure. Learning to read these messages is the fastest debugging skill you can build, because it tells you which of the other recipes in this chapter to reach for.

Common error patterns:

ERROR MESSAGE	LIKELY CAUSE
"Expected to find text X"	Page content not loaded yet
	or wrong page displayed
"Timeout 45000ms exceeded"	Whole step hit the Cucumber
	step timeout: page never
	loaded, or element never
	appeared
"Element not visible"	Element exists but hidden
	(display:none, behind modal)
"Strict mode violation"	Multiple elements match,
	Playwright expects one
"No visible modal found"	Modal did not open, or
	wrong selector
"Step undefined"	No matching step definition
	(check spelling, quotes)
"ECONNREFUSED"	Application is not running
	at the expected URL

How It Works: the message encodes which layer gave up. `ECONNREFUSED` is the network layer, the app is not running at `LAUNCH_URL`, so nothing browser-related matters yet. “Step undefined” is Cucumber’s matcher, your Gherkin text did not match any of the 411 built-in step expressions, usually a typo or a missing quote (Recipe 205 catches these in seconds). “Timeout 45000ms exceeded” is the Cucumber step timeout from the default profile, the step as a whole never finished, which usually means a navigation hung. A web-first assertion failure (“expected … to be visible”) means the framework already did the patient part for you: it polled every 100ms for the full 5-second window, through smartSettle’s AJAX/timer/DOM-quiet checks, and the element genuinely never reached that state. That last point changes your diagnosis: with webship-js 2.0 you can mostly stop asking “did the test wait long enough?” and start asking “why did the app never produce this?”, a much better question.

At scale: write this table into a one-page triage guide and put it in every project repository. The goal is that anyone on the team, including the engineer seeing this project for the first time, can classify a failure (app down / test broken / regression / flake) without summoning the person who wrote the suite. Cross-project on-call rotation only works when the failure taxonomy is shared.

Recipe 205 Using Dry-Run to Verify Step Matching

Problem: A product owner drafted ten new scenarios straight from the acceptance criteria, exactly what you want to encourage. But one step says `When I fill "Email" with` instead of `When I fill in "Email" with`, and you only find out twelve minutes into the run when the scenario dies as UNDEFINED. Worse: the failure makes the PO feel they “broke the tests”, which is the fastest way to make them stop writing scenarios.

Solution:

```
npx cucumber-js --dry-run

# Output shows each step's match status:
# ✓ Given I am on homepage                    ▶ matched
# ✓ When I go to "/login"                     ▶ matched
# ? When I do something custom                ▶ UNDEFINED (no match!)
```

FIGURE 22.1 *Using Dry-Run to Verify Step Matching (Recipe 205)*

This catches typos and missing step definitions before you waste time running tests.

How It Works: `--dry-run` loads everything a real run loads, the feature files, the 411 built-in step definitions, your custom steps from `tests/step-definitions/`, and runs the matching phase only. Every step's text is checked against the registered Cucumber expressions; no browser launches, no page loads, so a whole suite verifies in a couple of seconds. It catches the silent killers: a missing `in`, curly quotes pasted from a Word document, a custom step that exists on one project but not the one you copied the scenario into. It cannot catch wrong selectors or wrong expectations, only that every line of Gherkin has an implementation waiting for it.

At scale: make `--dry-run` the first job in every project's pipeline, it costs seconds and fails fast, before you pay for browser minutes. It is also your portability check: before promoting a feature file from one project into a shared library, dry-run it against each consuming project to prove every step resolves there too.

Recipe 206 Running Tests in Headed Mode

Problem: The error says the dashboard never appeared. You have re-read the scenario five times and it looks right. You are now debugging blind, reasoning about what a headless browser might have shown, instead of just looking. Ten minutes of guessing could have been thirty seconds of watching.

Solution:

```
# Watch the test run in a real browser window
HEADLESS=false npx cucumber-js tests/features/login.feature

# Slow it down so you can follow
HEADLESS=false SLOW_MO=2000 npx cucumber-js tests/features/login.feature
```

This is the single most effective debugging technique. You can see:

- Whether the page loaded
- Whether the login form appeared
- Whether the button was clicked
- What error messages the UI shows

How It Works: `HEADLESS=false` flips the Playwright launch option in `playwright.config.ts`, the exact same Chromium (or Firefox/WebKit via the `BROWSER` env var) runs, just with a window. webship-js also adapts its pacing: `SLOW_MO` defaults to 0 in CI, 300ms in headless local runs, and 800ms when headed, so a headed run is already watchable without any flags; set `SLOW_MO=2000` when you need to read each action. Nine times out of ten the bug is obvious within one viewing, a validation error under the form, a cookie-consent banner covering the button, a redirect to the wrong environment. The failure modes you cannot see in an error message are exactly the ones you can see through a window.

Pro Tip: combine headed mode with a single feature file path, as above. Watching one failing scenario is debugging; watching the whole suite is a movie you do not have time for.

At scale: headed mode is the one technique in this chapter that does not travel to CI, there is no window on a runner. The multi-project habit that replaces it: reproduce locally against the failing project's own environment by pointing `LAUNCH_URL` at it, set `BROWSER` to whatever the pipeline used, and watch the exact scenario that failed. Most "CI-only" failures stop being CI-only the moment the same scenario runs against the same environment with your eyes on it, and the ones that survive that test are genuinely about the runner, which is your platform team's lead, not yours.

Recipe 207 Taking Screenshots Mid-Test

Problem: The failure only happens in CI, where there is no window to watch. The After-failure screenshot shows you the crime scene, but not the lead-up, was the form ever filled? Did the page render before the click? You need the browser's state at specific moments along the way, not just at the end.

Solution:

```
Feature: Debug Login
  Scenario: Debug login failure
    Given I am on homepage
    When I go to "/login"
    And I take a screenshot named "01-login-page"
    And I fill in "Email" with "admin@example.com"
    And I fill in "Password" with "AdminPass123"
    And I take a screenshot named "02-form-filled"
    And I press "Sign In"
    And I take a screenshot named "03-after-login"
    Then I should see "Dashboard"
```

The screenshots create a visual timeline of the test execution. Examine them in `screenshots/` to see exactly what the browser showed at each point.

How It Works: you rarely need to hand-place screenshot steps, because webship-js captures automatically. `WEBSHIP_SCREENSHOT_ON_FAILED` is `true` by default, every failing scenario already leaves a `failed_`-prefixed image in `WEBSHIP_SCREENSHOT_DIR` (default `./screenshots`). For a full timeline, tag the scenario `@screenshots` and the framework captures after *every* step of that scenario, the tag version of `WEBSHIP_SCREENSHOT_ON_EVERY_STEP`, scoped to just the scenario you are investigating. File names follow `WEBSHIP_SCREENSHOT_PATTERN` (`{date-time}.{feature_file}.feature_{step_line}.{ext}` by default), so each image points back to the exact step line that produced it. The named-screenshot steps in the example are still the right tool when you want a stable, human-readable name to compare run over run.

At scale: keep the naming pattern identical across all projects and have CI archive the screenshots directory as a build artifact on failure. When a remote engineer triages another team's red build, "open artifacts, read filenames in order, find the step line" is a procedure that works on every project the same way, no tribal knowledge needed. Set `WEBSHIP_SCREENSHOT_PURGE` so a runner that survives between builds does not mix last night's images into today's investigation.

Recipe 208 Browser Console Log Capture

Problem: The button click does nothing, the assertion fails, and the screenshot shows a perfectly normal-looking page. What it does not show: the JavaScript exception that killed the click handler. In a headless CI run, console errors vanish into nowhere, the most informative line of evidence in the whole failure, and nobody saw it.

Solution: webship-js 2.0 captures page JavaScript errors out of the box, you configure it, not code it:

```
# Default: errors are reported as warnings alongside the scenario
WEBSHIP_JS_ERROR_MODE=warn npx cucumber-js

# Strict: any page JS error fails the scenario
WEBSHIP_JS_ERROR_MODE=fail npx cucumber-js

# Widen capture to warnings and info, ignore known third-party noise
WEBSHIP_JS_ERROR_LEVELS=error,warning \
WEBSHIP_JS_ERROR_IGNORE="analytics|gtag" \
npx cucumber-js
```

Per-scenario control via tags: `@js-fail` (or `@javascript`) makes errors fail that scenario, `@js-warn` downgrades to warnings, `@js-off` disables capture entirely.

If you also want non-error console output (logs, debug noise) attached to failures, add a custom hook:

```
// tests/support/hooks.js
const { Before, After } = require('@cucumber/cucumber');

Before(async function () {
  // After opening browser, set up console log capture
  this.consoleLogs = [];
  this.page.on('console', msg => {
    this.consoleLogs.push({
      type: msg.type(),
      text: msg.text(),
    });
  });
});

After(async function (scenario) {
  // Print console errors if test failed
  if (scenario.result.status === 'FAILED') {
    const errors = this.consoleLogs.filter(l => l.type === 'error');
    if (errors.length > 0) {
      console.log('\n--- Browser Console Errors ---');
      errors.forEach(e => console.log(`  [${e.type}] ${e.text}`));
      console.log('---');
    }
  }
});
```

How It Works: the built-in capture listens for page errors on every scenario and routes them according to `WEBSHIP_JS_ERROR_MODE`, `warn` (the default) surfaces them in the output without failing anything, `fail` turns the page's JavaScript health into an assertion, `off` silences it. `WEBSHIP_JS_ERROR_LEVELS` controls which console levels count (just `error` by default), and `WEBSHIP_JS_ERROR_IGNORE` takes a regex for the inevitable third-party noise, analytics beacons, ad scripts, browser-extension chatter, so one flaky CDN does not fail your suite. The tag overrides matter for precision: keep the suite in `warn` globally, then mark your most critical journeys `@js-fail` so a broken click handler on checkout is a failure, not a footnote.

At scale: standardize the env vars in your shared CI templates instead of per-project configs, one place to tighten `warn` to `fail` when a portfolio is ready, one shared `WEBSHIP_JS_ERROR_IGNORE` pattern for the marketing scripts every site carries. JS-error capture is also a free early-warning system: a new error appearing across several projects' runs on the same night usually means a shared dependency or CDN changed, and you have found it before any user reported it.

Recipe 209 Network Request Inspection

Problem: The UI swallows failures politely: the login form just sits there, no error toast, nothing in the screenshot. Meanwhile the API returned a 401 that the front end never surfaced. Without seeing the traffic, you will debug the form for an hour when the real story is one HTTP status code away.

Solution: for assertions inside a scenario, use the built-in recording steps:

```
Given I start recording network requests
When I press "Sign In"
Then a POST request to "**/api/login" should have been made
```

For automatic evidence on every failure, add a hook that logs error responses:

```
// tests/support/hooks.js
Before(async function () {
   this.networkRequests = [];
   this.page.on('response', response => {
      if (response.status() >= 400) {
         this.networkRequests.push({
            url: response.url(),
            status: response.status(),
            method: response.request().method(),
         });
      }
   });
});

After(async function (scenario) {
   if (scenario.result.status === 'FAILED' && this.networkRequests.length > 0) {
      console.log('\n--- Failed Network Requests ---');
      this.networkRequests.forEach(r =>
         console.log(`        ${r.method} ${r.url}         ▶ ${r.status}`)
      );
      console.log('---');
   }
});
```

FIGURE 22.2 *Network Request Inspection (Recipe 209)*

Output on failure:

```
--- Failed Network Requests ---
  POST http://localhost:8080/api/login → 401
  GET http://localhost:8080/api/dashboard → 403
---
```

That output reads like a confession: the login API rejected the credentials (401), so the dashboard refused access (403). The form was never the problem.

How It Works: the built-in steps and the custom hook tap the same source, Playwright's network events on the page. `Given I start recording network requests` begins collecting every request the browser makes; `Then a POST request to "**/api/login" should have been`

`made` matches the log against a glob pattern, the same pattern syntax `page.route()` uses for mocking. That makes it the assertion of choice for invisible behavior: analytics calls, autosaves, background syncs, things with no UI to assert on. The hook variant filters `response` events for status >= 400 and prints them only when the scenario fails, so passing runs stay quiet and failing runs carry their own network evidence.

At scale: ship the failed-request hook in your shared support package so every project's failure output includes it, it converts a whole class of cross-team tickets from "the test is flaky" to "your API returned 500 at 02:14, here is the URL". DevOps will notice the difference: failures that arrive with status codes and endpoints get routed to the right owner on first triage instead of bouncing through the QA channel.

Recipe 210 Video Recording of Test Runs

Problem: A scenario fails one night in five, only in CI, never locally. The screenshot shows the end state; what you need is the thirty seconds *before*, the redirect that happened, the spinner that never resolved, the modal that flashed and vanished. For intermittent CI failures, video is the evidence that ends the speculation.

Solution: video recording is built into webship-js 2.0, one environment variable, no hook code:

```
# Record only scenarios that fail (the right default for CI)
WEBSHIP_VIDEO=on-failure npx cucumber-js

# Record everything (debugging sessions)
WEBSHIP_VIDEO=on npx cucumber-js

# Record only scenarios tagged @video
WEBSHIP_VIDEO=tag npx cucumber-js

# Choose where videos land (default ./videos)
WEBSHIP_VIDEO=on-failure WEBSHIP_VIDEO_DIR=./artifacts/videos npx
cucumber-js
```

Per-scenario control in the feature file:

```
@video
Scenario: Checkout completes with a saved card
  # Always recorded, regardless of pass/fail

@no-video
Scenario: Bulk import of 5000 rows
  # Never recorded -- long scenario, huge file, no debugging value
```

How It Works: under the hood webship-js passes Playwright's `recordVideo` context option for you, 1280x720 WebM files, named by the pattern `{datetime}.{feature_file}.{scenario}.{status}.{ext}` so a directory listing already tells you which scenario failed and when. Playwright can only decide to record when the browser context is created, which is why this is a context-level setting and why `on-failure` mode records everything and discards the passes. It is also why the mid-scenario `start/stop video recording` steps come with a warning: starting recording mid-flight forces webship-js to rebuild the browser context, and the page state, cookies, session, the page you were on, is lost. Use the env var or tags instead unless you genuinely need a partial recording and your scenario re-establishes its own state afterward.

At scale: standardize on `WEBSHIP_VIDEO=on-failure` in the shared CI template and have the pipeline upload `WEBSHIP_VIDEO_DIR` as an artifact only when the job fails. Green runs cost you nothing; red runs come with a replay any engineer on any team can watch. Storage adds up across forty nightly suites, so set an artifact retention window, a video older than the next deploy has already done its job.

Debugging Quick Reference

TECHNIQUE	WHEN TO USE
HEADLESS=false	First thing to try
SLOW_MO=2000	Test runs too fast to follow
--dry-run	Step not matching
@screenshots tag	Visual timeline of a scenario
WEBSHIP_JS_ERROR_MODE=warn/fail	Suspect JS errors
Network request logging	Suspect API failures
WEBSHIP_VIDEO=on-failure	Intermittent failures in CI
print API response	API test returns wrong data
print current URL	Lost in a redirect chain
Screenshot on failure	On by default, keep it on

Chapter Summary

- Read error messages carefully, they indicate which layer gave up, and which recipe to reach for
- `--dry-run` catches undefined steps in seconds, before any browser launches
- `HEADLESS=false` is the most effective debugging technique; `SLOW_MO` makes it watchable
- The `@screenshots` tag turns one scenario into a visual timeline; failure screenshots are on by default
- `WEBSHIP_JS_ERROR_MODE` captures page JavaScript errors, `warn` globally, `@js-fail` on critical journeys
- Built-in network recording steps plus a failed-request hook turn API failures into evidence
- `WEBSHIP_VIDEO=on-failure` is the right CI default for intermittent failures
- Standardize all of it in shared CI templates so every project fails with the same evidence attached

Next: Test Maintenance, keeping your test suite healthy over time.

Chapter 23: Test Maintenance

Writing tests is the beginning. Maintaining them is the real work. Applications evolve: new features land, old features change, UIs get redesigned, APIs get versioned. Every change potentially affects your test suite. Without a maintenance discipline, the suite rots until it is more expensive to maintain than the bugs it catches.

And maintenance debt compounds across projects. A team that keeps one suite healthy by heroics will not keep twenty healthy the same way, the redesign that costs you an afternoon on one site costs a month across a portfolio, unless the suites share conventions, shared steps, and a review discipline. Every recipe in this chapter is written with that multiplication in mind.

Recipe 211 Refactoring Steps When the UI Changes

Problem: Monday morning, the redesigned login page ships. The “Sign In” button is now “Log In,” the “Email” field is now “Username,” and every suite that logs in, which is every suite, goes red at once. The team’s first instinct is to panic; yours should be to notice that this is the *good* failure mode: the tests are telling you exactly what changed, in the language users see.

Solution:

```
# Before the UI change:
When I fill in "Email" with "admin@example.com"
And I press "Sign In"

# After the UI change:
When I fill in "Username" with "admin@example.com"
And I press "Log In"
```

Approach:

1. Run the test suite, identify which tests fail
2. Update Gherkin steps to match the new UI labels
3. If you have custom step definitions with hardcoded selectors, update those
4. Run the suite again to verify all fixes

How It Works: the reason this refactor is a find-and-replace instead of an archaeology dig is web-ship-js's selector strategy. `I fill in "Email"` resolves label-first, label text, then id, name, placeholder, and `I press "Sign In"` finds the button by its accessible name through `getByRole`. The test's contract is the words on the screen, so when the words change, the *Gherkin* changes, a diff any product owner can read and approve, while the step definitions stay untouched. Compare that with a selector-based suite, where the same redesign means hunting CSS classes through support code. The depth of the change matches the depth of what changed: copy change, copy-level fix.

At scale: centralize before the redesign, not after. If login lives in a `Background:` or one shared step per project, the rename is one edit per repo; if forty feature files each spell out the login form, it is forty edits times twenty projects. Where you do need raw selectors, keep them in named-selector JSON files under `tests/selectors/` so a markup change is one JSON edit. The redesign that hits a prepared portfolio is an afternoon; the same redesign on an unprepared one is the sprint nobody planned for.

Recipe 212 Updating Tests for New Features

Problem: Security ships two-factor authentication on a Thursday. It is a genuine improvement, and it just inserted a step into the middle of every logged-in scenario you own. Two hundred scenarios across a dozen features now stall at a verification prompt none of them know exists. The wrong fix is editing two hundred scenarios; the right fix is editing the one step they all share.

Solution:

```
// Add a new step for the 2FA code
When('I enter the 2FA code', async function () {
  // In test environment, the 2FA code is always '123456'
  await this.page.getByLabel('Verification Code').fill('123456');
  await this.page.getByRole('button', { name: 'Verify' }).click();
  await this.page.waitForLoadState('networkidle');
});
```

Update the login step that many tests depend on:

```
Given('I am a logged in user with the {string} user', async function
(role) {
  const user = users[role];
  await this.page.goto(this.launchUrl + '/login');
  await this.page.waitForLoadState('networkidle');
  await this.page.getByLabel('Email').fill(user.email);
  await this.page.getByLabel('Password').fill(user.password);
  await this.page.getByRole('button', { name: 'Sign In' }).click();
  await this.page.waitForLoadState('networkidle');

  // Handle 2FA if present
  const twoFaInput = this.page.getByLabel('Verification Code');
  if (await twoFaInput.isVisible({ timeout: 2000 }).catch(() => false)) {
    await twoFaInput.fill('123456');
    await this.page.getByRole('button', { name: 'Verify' }).click();
    await this.page.waitForLoadState('networkidle');
  }
});
```

Now all tests that use `Given I am a logged in user` continue to work without changes.

How It Works: this is the payoff of funneling shared behavior through one step definition: the 2FA handling lands in one place, behind the same Gherkin sentence two hundred scenarios already speak. The conditional check (`isVisible` with a short timeout, swallowing the miss) makes the step tolerant of both worlds, environments where 2FA is enabled and environments where it is not yet, which matters mid-rollout when staging has the feature and the demo site does not. The hard-coded `123456` is the standard trick of pinning the verification code in the test environment; the alternative, scraping real codes from email or TOTP, buys realism you rarely need at the cost of flake you definitely do not.

At scale: logging in through the form in every scenario is also a speed and stability tax that grows with feature complexity, 2FA just made every scenario two steps slower. webship-js ships auth-state steps that save and restore the browser's `storageState`: log in once, save the state, and let subsequent scenarios restore it instead of re-running the form. Across a portfolio, pair that with one rule: every project uses the same login step sentence. Then a platform-wide auth change (SSO migration, CAPTCHA, new 2FA flow) is one step-definition edit per project, or one edit in a shared step package, instead of an all-hands week.

Recipe 213 Removing Obsolete Tests

Problem: The wishlist feature was retired two quarters ago, but its scenarios still run nightly, failing, then skipped, then tagged `@flaky`, then quietly ignored. Now they are noise in every report, minutes in every CI run, and a small lie in your coverage numbers. Dead tests are worse than no tests: they train the team to ignore red.

Checklist:

```
Before removing a test, verify:
  ☐ The feature was intentionally removed (not just temporarily hidden)
  ☐ The test is not covering behavior that still exists elsewhere
  ☐ The team agrees the test should be removed (PR review)
  ☐ The feature file deletion is in its own commit (easy to revert)
```

Process:

1. Run the test, if it fails with “page not found” or “element not found,” the feature is gone
2. Check with the product owner: “Is this feature permanently removed?”
3. Delete the feature file
4. Delete any custom step definitions that were ONLY used by that feature
5. Run the remaining test suite to ensure nothing else depended on the deleted steps

How It Works: the product-owner check in step 2 is the load-bearing one, “hidden behind a feature flag for one client” and “removed” look identical to a failing test, and only the PO knows which it is. This is a place where readable Gherkin earns its keep: you can paste the scenario into the question and get a real answer. Step 5 is cheap insurance done right: after deleting steps, run `npx cucumber-js --dry-run`, it verifies in seconds that every remaining step still has a definition, without burning a full browser run. Keeping the deletion in its own commit means that when the feature comes back next year (features do), `git revert` resurrects the tests intact.

At scale: schedule pruning instead of waiting for it to hurt, a quarterly pass per project, looking for scenarios that have not changed status in months and tags that no pipeline selects anymore. Track scenario count alongside pass rate in your cross-project reporting: a suite that only ever grows is not a healthier suite, it is a slower one. Deleting a dead test is a maintenance win worth the same review ceremony as adding a live one.

Recipe 214 Managing Test Data Over Time

Problem: The suite was born against a database where “testuser1” existed and the catalog had exactly twelve products. A year later, a schema migration renamed a column, someone cleaned up staging, and a marketing import added four hundred products. None of your code changed, but a third of the suite fails, and the new QA engineer spends her first week discovering that the tests do not document what data they expect. Stale data is the slowest-burning form of maintenance debt because nothing announces it.

Solution:

Test Data Management Strategies:

Strategy 1: Create data per test (recommended)
Each test creates what it needs via API or UI
Pros: Independent, no stale data, runs anywhere
Cons: Slower setup per test

Strategy 2: Seed script before suite
A script runs before all tests to set up known state
Pros: Fast test execution, known state
Cons: Seed script must be maintained, can get out of sync

`Strategy 3: Database snapshot/restore`
Snapshot a clean database, restore before each test run
Pros: Fastest, always consistent
Cons: Snapshot must be updated when schema changes

FIGURE 23.1 *Managing Test Data Over Time (Recipe 214)*

How It Works: the strategies differ in *where the truth about test data lives*. With per-test creation, the truth is in the scenario itself, `Given there is a user named "Sara"` both documents and provisions, usually through the API steps (`When I send a POST request to "/api/users" with values:`), which cost milliseconds instead of UI seconds. With a seed script, the truth lives in one versioned script that drifts unless someone owns it. With snapshots, the truth is a binary blob nobody can read in review. That is why the recommendation is per-test creation by default, with seeds or snapshots reserved for the expensive substrate, roles, taxonomies, configuration, that every scenario shares and no scenario should rebuild. Mixed mode is normal: seed the skeleton, create the flesh per scenario.

At scale: per-test creation is also what makes one suite pattern portable across twenty projects, because it removes "prepare the database" from every onboarding checklist, point `LAUNCH_URL` at any environment and run. This was the working rule for testing the Varbase distribution across

many sites: the installation profile provides the same baseline everywhere, and every scenario builds its own content on top. If a scenario only passes on one specially-groomed environment, treat that as a defect in the scenario.

Recipe 215 Code Review Checklist for Tests

Problem: Application code gets two reviewers and a style bot; test code gets "LGTM, it's green". Then the new hire's first PR lands with a `wait 5s`, a hardcoded staging URL, and a step definition that duplicates one three files away, and it merges, because nobody knew what to look for. Six months of that is how a suite rots while every individual PR looked fine.

When reviewing a PR that includes test changes:

```
Test PR Review Checklist:
  ☐ Scenarios describe behavior, not implementation
  ☐ Step text uses business language
  ☐ No hardcoded wait times where smart waits would work
  ☐ No shared state between scenarios
  ☐ Tests create their own data (no fixture dependencies)
  ☐ Assertions check both positive and negative cases
  ☐ Tags are consistent with project conventions
  ☐ New step definitions do not duplicate existing ones
  ☐ Error messages in custom assertions are descriptive
  ☐ No console.log or debug statements left in
```

How It Works: every line of the checklist encodes one of the failure modes from Chapter 21, caught at the cheapest possible moment, before merge. "No hardcoded wait times" blocks the static sleeps that smartSettle exists to replace. "Tests create their own data" blocks the fixture coupling that breaks the third environment you deploy to. "New step definitions do not duplicate existing ones" is the big one on a mature project: with 411 built-in steps plus your custom ones, the odds are good the step already exists, a quick `--dry-run`, or a grep through `tests/step-definitions/`, beats writing a near-twin that drifts apart from the original. The repo's own rule is worth adopting too: every new step definition gets documented with at least five example lines, so the next reviewer can tell at a glance what it is for.

At scale: make the checklist a PR template in every repository, the same items, the same order, across all twenty projects. That does two things: review quality stops depending on which senior happens to be on rotation, and engineers who move between projects already know the bar. For a QA lead, the checklist is also a teaching document, the new hire's first PR gets reviewed against written criteria instead of folklore, and the feedback points at a line in the template, not at a person's opinion.

Chapter Summary

- Test maintenance is an ongoing discipline, not a one-time task, and it compounds across projects
- UI label changes require Gherkin updates, not step definition changes, label-first and role-based selectors make it so
- New features may require updating shared steps (like login); shared steps are why that is cheap
- Remove obsolete tests deliberately, not accidentally, and verify with `--dry-run` after deleting steps
- Manage test data through per-test creation by default; seed only the shared substrate
- Use the same code review checklist for test PRs across every project

Next: Performance Optimization, making your test suite faster.

Chapter 24: Performance Optimization

A fast test suite gets run. A slow test suite gets ignored. When your suite takes forty-five minutes, developers push code without waiting for it. When it takes five minutes, it becomes a natural part of the workflow.

And on a multi-project team, slow is not just a culture problem, it is a bill. Forty suites times forty-five minutes is thirty hours of runner time a night, paid in CI minutes or in queued pipelines. Every minute you shave off the suite pattern gets multiplied by every project that uses it.

This chapter shows you how to measure, identify bottlenecks, and optimize your test suite's performance.

Recipe 216 Measuring Test Suite Performance

Problem: The suite "feels slow", so someone proposes buying bigger runners, someone else blames the staging server, and a third person wants to delete tests. Nobody has a number. Optimizing without measuring is how teams spend a sprint making the wrong thing 10% faster while three scenarios quietly burn four minutes each.

Solution:

```
# Time the entire suite
time npx cucumber-js

# Use the JSON formatter to get per-scenario timing
npx cucumber-js --format json:report.json

# Parse the report for slowest scenarios
node -e "
  const report = require('./report.json');
  const scenarios = [];
  report.forEach(feature => {
    feature.elements.forEach(scenario => {
      const duration = scenario.steps.reduce((sum, s) =>
        sum + (s.result?.duration || 0), 0);
      scenarios.push({
        name: scenario.name,
        duration: (duration / 1e9).toFixed(2) + 's',
        ms: duration / 1e6,
      });
    });
  });
  scenarios.sort((a, b) => b.ms - a.ms);
  console.log('Slowest scenarios:');
  scenarios.slice(0, 10).forEach((s, i) =>
    console.log(\`  \${i+1}. \${s.duration}  \${s.name}\`)
  );
"
```

How It Works: Cucumber's JSON formatter records a nanosecond duration on every step result, and the snippet above just sums them per scenario and sorts. webship-js already writes this report on every run, the default profile sends JSON to `tests/reports/cucumber_report.json`, so on most projects you can point the script at the file you already have instead of re-running anything. Suite time almost always follows a power law: the top ten scenarios carry a third of the runtime. That top-ten list is your work queue for the rest of this chapter, and it tells you which fix applies, because a 90-second scenario is usually either a UI journey that belongs at the API level (Recipe 218) or a login repeated through the form forty times.

At scale: the same JSON feeds `bin/generate-reports.js`, which stamps reports with `--app-version`, `--env`, `--browser`, `--executed`, and arbitrary `--metadata Key=Value` pairs. Add a project identifier and collect the JSON files centrally, and you get the chart your delivery managers actually want: suite duration per project, per night, trending over a quarter. A suite that doubled its runtime in three months is a maintenance conversation you want to have *before* the nightly window overflows, and the data makes it a planning item instead of an argument.

Recipe 217 Optimizing Wait Times

Problem: The quarterly platform review lands and CI minutes are up forty percent, the first proposal on the table is bigger runners. Before anyone spends money, profile a suite; the verdict is almost always the same: most of the wall-clock time is nobody doing anything. The test is waiting on the app, or worse, waiting on a number someone typed two years ago. Claim back the free seconds first, the delays that exist for humans watching the screen, not for correctness.

Audit your waits:

WAIT TYPE	WHERE IT COMES FROM	CAN OPTIMIZE?
smartSettle (navigation)	3000ms budget per page load	No, exits the
smartSettle (after step)	1500ms budget per step	No, same early exit
web-first assertions	poll 100ms, up to 5s	Only cost time when
minWaitTime.page	worldParameters (3000ms)	Lower only if the
before/after step delays	worldParameters minWaitTime	Set to 0 in CI
SLOW_MO	0 CI / 300 headless /	Keep 0 in CI; it
wait Ns (static sleeps)	legacy suites	Delete, never

Optimize cucumber.js for CI:

```
// cucumber.js - CI profile
module.exports = {
  ci: {
    worldParameters: {
      launchUrl: process.env.LAUNCH_URL || 'http://localhost:8080',
      minWaitTime: {
        before_scenario: 0,   // No delay
        after_scenario: 0,    // No delay
        before_step: 0,       // No delay
        after_step: 0,        // No delay
        page: 3000,           // Page load timeout only
      },
    },
  },
};
```

```
# CI run: no unnecessary delays
SLOW_MO=0 npx cucumber-js --profile ci
```

How It Works: the key distinction is budgets versus delays. smartSettle's numbers, 3000ms per navigation, 1500ms after each state-changing step, are *budgets*: it polls the AJAX counter, pending timers, and DOM-mutation timestamp every 50ms and returns the instant the page is quiet, so a fast page pays 300ms, not 3000ms. Cutting those buys you nothing on a healthy app and flake on a slow one. The `minWaitTime` values and `SLOW_MO`, by contrast, are *delays*, unconditional pauses that exist so a human can watch, and CI has no humans, so they go to zero. webship-js already defaults `SLOW_MO` to 0 in CI; the profile above makes the rest explicit. One thing not to do: do not reach for `WEBSHIP_AUTO_SETTLE=off` to save milliseconds. The settle cycle is what lets the suite run without a single static sleep, switching it off trades a rounding error of speed for the entire flake category Chapter 21 exists to fix.

At scale: keep one `ci` profile shape in a shared template and configure differences through env vars (`LAUNCH_URL`, `BROWSER`, `SLOW_MO`) rather than editing `cucumber.js` per project. The day someone discovers a better timing setting, it rolls out by updating the template, not by twenty pull requests, of which seventeen get merged and three are forgotten until their pipelines mysteriously run slow.

Recipe 218 Moving Tests to the Right Level

Problem: Someone once wrote “invalid email shows an error” as a browser test, and it became the template. Now there are thirty validation scenarios, each one loading the registration page, filling the form, submitting, and waiting for the render, twenty seconds apiece to verify decisions the server makes in milliseconds. Ten minutes of every run is spent driving a browser to ask questions that never needed one.

GO TO /REGISTER	POST /API/USERS
Fill in name	with invalid
Fill in email	email
Press submit	Assert 422

Rule of thumb: If the test does not need to verify visual behavior or user workflow, use an API test.

How It Works: the API steps live in the same webship-js suite and the same report, `Given the API base URL is "..."`, `When I send a POST request to "/api/users" with values:`, `Then the API response code should be 422`, `Then the API response should contain "valid email"`. Under the hood they are axios calls: no browser launch, no page render, no settle cycle, which is where the 20s-to-0.5s drop comes from. Migrate by class, not by scenario: all field validation moves down, all permission-by-role checks move down, and you *keep one* browser scenario per form to prove the error actually renders where a user can see it. The server saying 422 and the user seeing “Please enter a valid email” are two different behaviors; test each once, at its own level.

At scale: this is the cheapest CI capacity you will ever add. Thirty validation scenarios going from browser to API saves roughly ten minutes per project per run, across twenty projects running nightly, that is more than three hours of runner time a night, bought with a refactor instead of a hardware budget. It also splits your reports along an ownership line a PMO can use: API failures mean the contract broke, browser failures mean the experience broke.

Recipe 219 Parallel Execution Configuration

Problem: Your CI runner has eight cores, and your suite uses one of them, scenario after scenario, single file, while the machine idles at 12%. The platform team sees a 40-minute pipeline and assumes the tests are heavy; the truth is they are sequential. Parallelism is the one optimization that needs no test changes at all, *if* your scenarios are isolated.

Solution:

```
# Detect available CPU cores and use them
npx cucumber-js --parallel $(nproc)

# Or set a fixed number
npx cucumber-js --parallel 4
```

package.json scripts:

```
{
  "scripts": {
    "test": "cucumber-js",
    "test:fast": "SLOW_MO=0 cucumber-js --parallel 4",
    "test:ci": "SLOW_MO=0 cucumber-js --parallel $(nproc) --profile ci"
  }
}
```

Requirements:

- Each scenario must be independent (no shared state)
- webship-js ensures this: each scenario gets its own browser
- Database operations should not conflict between parallel scenarios

How It Works: `--parallel N` makes Cucumber start N worker processes and hand scenarios to whichever worker frees up first. Each webship-js worker runs its own browser with its own context, so cookies, sessions, and localStorage are isolated by construction, the only sharing left is whatever your scenarios do to the application's database, which is why Recipe 194's isolation work is the real prerequisite. Mind the memory math when picking N: every worker is a full

browser, so `$(nproc)` on a 16-core runner can mean sixteen Chromium instances and an out-of-memory kill that looks like random flake. Start at 4, watch runner memory and the application server's response times, and raise it until one of them objects. And if going parallel surfaces failures, do not turn it off, it just found real isolation bugs that sequential luck was hiding.

At scale: parallelism moves the bottleneck from the runner to the system under test, twenty workers across several projects hammering one shared staging server will manufacture timeouts that read as test flake but are capacity. Coordinate with your platform team on the nightly picture: workers per suite times suites per window equals concurrent browser sessions staging must absorb. On metered CI, also note what parallel does and does not save: it cuts wall-clock minutes per pipeline (and developer waiting), while total compute stays roughly the same, the win is feedback speed and nightly-window fit, not a smaller invoice.

Performance Optimization Checklist

```
Quick Wins (minutes to implement):
  ☐ Set SLOW_MO=0 in CI
  ☐ Set all minWaitTime values to 0 in CI profile
  ☐ Remove unnecessary "wait Ns" steps
  ☐ Enable --parallel 4 (or more)

Medium Effort (hours):
  ☐ Convert data validation tests from browser to API
  ☐ Delete legacy "wait Ns" steps -- smartSettle and web-first
    assertions already cover them
  ☐ Remove redundant scenarios that test the same path
  ☐ Combine related assertions into fewer scenarios

Architecture Changes (days):
  ☐ Implement test data creation via API (faster than UI)
  ☐ Set up CI matrix for parallel browser testing
  ☐ Log in once and reuse it via the storage-state
    save/restore steps
  ☐ Build a smoke test suite for quick PR feedback
```

Chapter Summary

- Measure before optimizing, the JSON report you already generate knows where the time goes
- Human-facing delays (`SLOW_MO`, minWaitTime) go to 0 in CI; smartSettle budgets stay, they exit early on their own
- Delete legacy static sleeps; never disable auto-settle to chase milliseconds
- Move data validation tests from browser to API level, same suite, same report, 40x faster
- Parallel execution can cut wall-clock time by 75%+, if scenarios are isolated
- Saving and restoring storage state beats logging in through the form in every scenario
- Smoke test suite provides fast PR feedback
- A fast test suite gets used; a slow one gets ignored, and across twenty projects, a slow one gets expensive

This concludes Part V: Troubleshooting and Maintenance.

Next: CI/CD Integration, putting your test suite into production pipelines.

Part VI: Real World

Chapter 25: CI/CD Integration

A test suite that only runs on your laptop is a hobby project. A test suite that runs automatically on every push, every pull request, and every deployment is a production-grade quality gate. CI/CD integration transforms your tests from “something I run sometimes” to “something that protects the team always.”

This chapter provides copy-paste CI configurations for the platforms you are most likely to meet. You will not be starting from scratch either way: the webship-js repo ships ready-made configs for thirteen CI providers, GitHub Actions, GitLab CI, Bitbucket Pipelines, Azure Pipelines, CircleCI, Travis, Jenkins, Drone, Semaphore, AWS CodeBuild, Google Cloud Build, Codefresh, and Bamboo. The recipes here explain the moving parts those configs share, because the day a pipeline fails, you need to know which part moved.

CI/CD Pipeline with Testing Gates

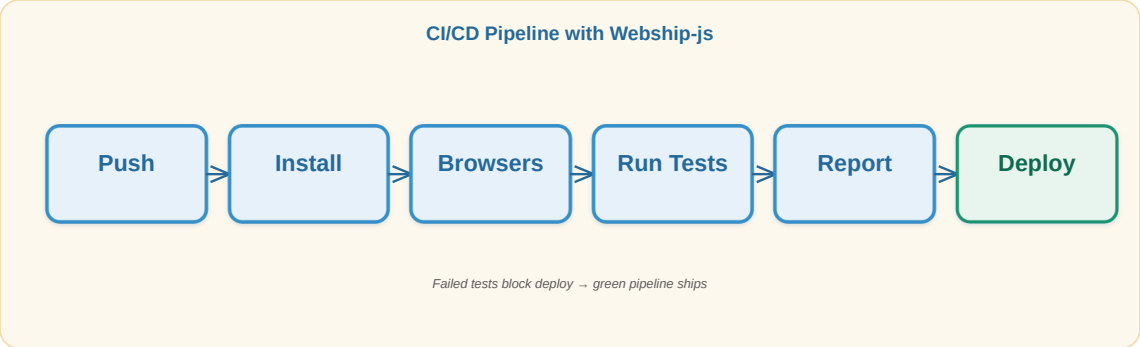


FIGURE 25.1 *CI/CD Pipeline with Testing Gates*

Recipe 220 GitHub Actions Setup

Problem: The suite is green on every laptop, and that is exactly the problem, it only runs when someone remembers to run it. A teammate merges late on a Thursday without running the tests, and Friday's standup opens with "who broke login?" Nobody broke it on purpose; there was simply no gate. You need the suite to run on every push and every pull request, with the results attached to the PR where the reviewer is already looking.

Solution:

```
# .github/workflows/test.yml
name: Test Suite

on:
  push:
    branches: [main, develop]
  pull_request:
    branches: [main]

jobs:
```

```
  smoke-test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: '20'
          cache: 'npm'

      - name: Install dependencies
        run: npm ci

      - name: Install Playwright browsers
        run: npx playwright install --with-deps chromium

      - name: Start application
        run: npm run start &
        env:
          NODE_ENV: test

      - name: Wait for application
        run: npx wait-on http://localhost:8080 --timeout 30000

      - name: Run smoke tests
        run: BROWSER=chromium SLOW_MO=0 npx cucumber-js --tags "@smoke"
        env:
          LAUNCH_URL: http://localhost:8080

      - name: Upload screenshots on failure
        if: failure()
        uses: actions/upload-artifact@v4
        with:
          name: screenshots
          path: screenshots/

  full-regression:
    needs: smoke-test
    runs-on: ubuntu-latest
    strategy:
      matrix:
        browser: [chromium, firefox, webkit]
```

```
    steps:
      - uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: '20'
          cache: 'npm'

      - name: Install dependencies
        run: npm ci

      - name: Install Playwright browsers
        run: npx playwright install --with-deps ${{ matrix.browser }}

      - name: Start application
        run: npm run start &
        env:
          NODE_ENV: test

      - name: Wait for application
        run: npx wait-on http://localhost:8080 --timeout 30000

      - name: Run tests on ${{ matrix.browser }}
        run: BROWSER=${{ matrix.browser }} SLOW_MO=0 npx cucumber-js
        env:
          LAUNCH_URL: http://localhost:8080

      - name: Upload test artifacts
        if: always()
        uses: actions/upload-artifact@v4
        with:
          name: test-results-${{ matrix.browser }}
          path: |
            screenshots/
            reports/
```

How It Works: The workflow defines two jobs with different speed budgets. `smoke-test` is the PR gate: it installs dependencies with `npm ci` (lockfile-exact, so CI runs the same dependency tree that passed on a laptop), installs Chromium plus its OS libraries with `--with-deps`, boots the app, and runs only `@smoke` scenarios on a single browser. `full-regression` runs after it, fanned out by the matrix into three independent runners, one per browser engine. The wiring be-

tween CI and webship-js is all environment variables, and that is deliberate. `BROWSER` is read by webship-js's `playwright.config.ts` to pick chromium, firefox, or webkit. `LAUNCH_URL` feeds `worldParameters.launchUrl`, so feature files never hardcode a host. `SLOW_MO=0` is technically redundant, webship-js already defaults slow-motion to 0 in CI, but stating it makes the config self-documenting. The `wait-on` step matters more than it looks: `npm run start &` returns immediately, and without the wait your first scenario races the app boot and dies with a connection error that has nothing to do with your tests. Finally, the screenshot upload runs `if: failure()`, webship-js captures a screenshot of every failed step on its own (`WEBSHIP_SCREENSHOT_ON_FAILED` defaults to true, files land in `./screenshots` with a `failed_` prefix), so the artifact holds exactly what you need on a red run and the step is skipped on green ones.

At scale: When you own twenty repos, do not paste this file twenty times. Extract it into a reusable workflow (`workflow_call`) in a shared repo and have each project call it with two inputs: the launch URL and the tag expression. A pipeline fix then ships to the whole fleet in one PR instead of twenty. Wire `LAUNCH_URL` through repository or environment variables instead of editing YAML per project, and budget the matrix consciously, three browsers times forty projects is 120 runner jobs a night. Most teams keep the PR gate on Chromium-only `@smoke` and spend the multi-browser minutes on a nightly schedule.

Pro Tip: Add `WEBSHIP_VIDEO: on-failure` to the regression job's `env` and include `videos/` in the artifact paths. A video of the failing scenario answers "what actually happened on the runner?" faster than any log, and green runs keep nothing, so artifact storage stays flat.

Recipe 221 GitLab CI Setup

Problem: The pipeline worked for months, then a routine `npm update` bumps Playwright and every job starts failing with "Executable doesn't exist", the runner image still carries last quarter's browsers. Or it is day one: tests pass locally and die in CI with missing system libraries, and you lose an afternoon to apt-get archaeology. Both are the same disease. The browser environment in CI is not the one your dependency tree expects.

Solution:

```
# .gitlab-ci.yml
image: mcr.microsoft.com/playwright:v1.58.0-jammy

stages:
  - test

variables:
  LAUNCH_URL: http://localhost:8080
  SLOW_MO: "0"

smoke-test:
  stage: test
  script:
    - npm ci
    - npm run start &
    - npx wait-on $LAUNCH_URL --timeout 30000
    - BROWSER=chromium npx cucumber-js --tags "@smoke"
  artifacts:
    when: on_failure
    paths:
      - screenshots/

full-regression:
  stage: test
  parallel:
    matrix:
      - BROWSER: [chromium, firefox, webkit]
  script:
    - npm ci
    - npm run start &
    - npx wait-on $LAUNCH_URL --timeout 30000
    - npx cucumber-js
  artifacts:
    when: always
    paths:
      - screenshots/
      - reports/
```

How It Works: The whole recipe hangs on the first line. `mcr.microsoft.com/playwright:v1.58.0-jammy` is Microsoft's official image with Chromium, Firefox, and WebKit preinstalled, along with every OS library they need, which is why there is no `playwright install` step anywhere in this file. The rule that keeps it working: the image tag must match the Playwright version in your `package.json`. Playwright looks for browser builds keyed to its own version, so if the lockfile says 1.59 and the image says v1.58.0, the builds it wants are simply not there, and you are back to "Executable doesn't exist". Treat the image tag and the dependency as one thing; upgrade them in the same merge request. The rest is wiring: `variables` exports `LAUNCH_URL` and `SLOW_MO` to every job, `parallel: matrix` expands `full-regression` into three jobs that each receive a different `BROWSER` value for webship-js's `playwright.config.ts` to read, and the two artifacts blocks differ on purpose, `when: on_failure` keeps smoke artifacts lean, while `when: always` preserves regression reports even on green runs so you have a baseline to compare against.

At scale: Forty projects means forty `.gitlab-ci.yml` files quietly pinning forty image tags, and they will drift. Move the image line and the job templates into a shared CI include (`include: project: qa/ci-templates`) and let each project's file shrink to a few variables. The next Playwright upgrade becomes one MR in one repo, and the fleet follows. It also gives your platform team a single answer to "which browser versions are we actually testing with?", a question that is otherwise forty answers.

Recipe 222 Jenkins Pipeline Setup

Problem: Your Jenkins agents are snowflakes. Someone installed Chrome on agent-03 by hand two years ago, agent-07 has a different glibc, and the same suite passes or fails depending on which agent grabs the build. Nobody remembers how the working agent was set up, so nobody dares touch it. You need builds that carry their browser environment with them.

Solution:

```
// Jenkinsfile
pipeline {
    agent {
```

```
        docker {
            image 'mcr.microsoft.com/playwright:v1.58.0-jammy'
        }
    }

    environment {
        LAUNCH_URL = 'http://localhost:8080'
        SLOW_MO = '0'
    }

    stages {
        stage('Install') {
            steps {
                sh 'npm ci'
            }
        }

        stage('Smoke Tests') {
            steps {
                sh 'npm run start &'
                sh 'npx wait-on $LAUNCH_URL --timeout 30000'
                sh 'BROWSER=chromium npx cucumber-js --tags "@smoke"'
            }
        }

        stage('Full Regression') {
            parallel {
                stage('Chromium') {
                    steps {
                        sh 'BROWSER=chromium npx cucumber-js'
                    }
                }
                stage('Firefox') {
                    steps {
                        sh 'BROWSER=firefox npx cucumber-js'
                    }
                }
                stage('WebKit') {
                    steps {
                        sh 'BROWSER=webkit npx cucumber-js'
                    }
                }
            }
```

```
        }
    }

    post {
        always {
            archiveArtifacts artifacts: 'screenshots/**,reports/**',
allowEmptyArchive: true
        }
    }
}
```

How It Works: `agent { docker { image '...' } }` is the snowflake killer: every build runs inside the pinned Playwright container no matter which physical agent picks it up, so "works on agent-03" stops being a sentence anyone says. The same version rule from the GitLab recipe applies, the image tag and your `package.json` Playwright version move together. The `environment` block exports `LAUNCH_URL` and `SLOW_MO` to every stage. One thing to watch: the three regression stages run in `parallel` inside the same container, which means three browsers share one agent's CPU. On a two-core agent they starve each other, scenarios stretch, and you can blow through the 45-second step timeout that webship-js's default Cucumber profile sets, failures that look like app bugs but are really contention. If you see that, run the browsers sequentially or give the job a bigger agent. The `post { always }` block archives screenshots and reports before the container is torn down; without it, the evidence vanishes with the workspace.

Pro Tip: If the app you started with `npm run start &` is mysteriously dead by the regression stage, you have met Jenkins's ProcessTreeKiller, which reaps background processes when a step ends. Start the app with `JENKINS_NODE_COOKIE=dontKillMe npm run start &` to keep it alive across stages.

Recipe 223 Test Reports in CI

Problem: The nightly run went red and the only record is a four-thousand-line console log on a CI page half the team cannot even access. The delivery manager asks “what failed, and does it block the release?”, and producing an honest answer takes you twenty minutes of scrolling. Results that exist only as terminal output are results only one person can read, and that person becomes the bottleneck for every release question.

Solution:

```
// cucumber.js - CI profile with JSON and HTML reporting
module.exports = {
  ci: {
    format: [
      '@cucumber/pretty-formatter',
      'json:reports/cucumber-report.json',
    ],
    // ... other config
  },
};
```

Generate HTML report after tests:

```
// generate-report.js
const reporter = require('cucumber-html-reporter');

reporter.generate({
  theme: 'bootstrap',
  jsonFile: 'reports/cucumber-report.json',
  output: 'reports/cucumber-report.html',
  reportSuiteAsScenarios: true,
  launchReport: false,
  metadata: {
    'Browser': process.env.BROWSER || 'chromium',
    'Platform': process.platform,
    'Environment': process.env.NODE_ENV || 'test',
  },
});
```

How It Works: Two formats, two audiences. The `ci` profile keeps the pretty formatter for humans watching the console and adds `json:reports/cucumber-report.json` for machines, a complete record of every scenario, step, status, duration, and attachment. `cucumber-html-reporter` then renders that JSON into a self-contained HTML page where a red scenario is a red row anyone can click open. The `metadata` block is not decoration: stamping browser, platform, and environment onto the report is what lets you tell "failed on webkit against staging" from "failed on chromium against prod" three weeks later. And webship-js ships this whole pipeline ready-made, `bin/generate-reports.js` wraps the same reporter with four themes (bootstrap, hierarchy, foundation, simple), can emit PDF as well as HTML (`--format html|pdf|all`, plus `--pdf-format` and `--pdf-landscape`), and takes metadata as flags: `--app-version`, `--env`, `--browser`, `--platform`, `--parallel`, `--executed`, and any number of custom `--metadata Key=Value` pairs. In CI you can pass extra flags through the `WEBSHIP_REPORT_ARGS` environment variable, or set `WEBSHIP_REPORT_DISABLE` for runs where you genuinely do not want a report generated.

At scale: The JSON report schema is identical for every project, and that is the lever. Stamp each run with `--app-version`, `--env`, and a `--metadata Project=name` pair, collect the JSON files from all forty nightly runs into one place, and a small script can answer "which projects went red last night, on which browsers, since which app version", the cross-project quality signal your PMO keeps asking for and nobody can currently produce. The PDF output earns its keep here too: a generated, versioned PDF attached to the release ticket is a sign-off artifact a non-technical stakeholder can actually open.

Recipe 224 Deployment Gate Testing

Problem: It is 4:45 on a Friday and the deploy job runs anyway, red suite, green deploy, because nothing in the pipeline actually connects them. The hotfix happens at midnight, and the retrospective writes itself: tests that do not gate anything are advisory, and advisory quality checks lose every argument with a deadline. The gate has to be structural, not a team agreement to "check the tests first".

Solution (GitHub Actions):

```
# Branch protection rule: Require "smoke-test" job to pass
# Settings ▶ Branches ▶ Branch protection rules ▶ Require status checks

# The workflow ensures tests must pass before merge:
jobs:
  smoke-test:
    # ... (must pass for PR merge)

  deploy:
    needs: [smoke-test, full-regression]
    if: github.ref == 'refs/heads/main'
    runs-on: ubuntu-latest
    steps:
      - name: Deploy to production
        run: echo "Deploying..."
        # Only runs if ALL test jobs passed
```

FIGURE 25.2 *Deployment Gate Testing (Recipe 224)*

How It Works: There are two separate gates here, and you want both. The merge gate is the branch protection rule: marking `smoke-test` as a required status check greys out the merge button until it passes, so red code cannot reach `main` by accident, not even with an approving review. The deploy gate is the `needs: [smoke-test, full-regression]` line: GitHub Actions builds a dependency graph from `needs`, and if any listed job fails, the deploy job is skipped, not failed, skipped, which reads honestly in the run timeline. The `if: github.ref == 'refs/heads/main'` condition keeps deploys off feature branches even though the same workflow runs there. For production you can add a third, human gate: a GitHub environment with required reviewers, so the pipeline proves the build is safe and a person still decides when it ships.

At scale: Use the same job names, `smoke-test`, `full-regression`, `deploy`, in every repository, because the names are the interface. A release manager who can read one project's checks can now read all forty, and "is release X safe?" stops being a meeting and becomes a glance at the gate. Pair that with the report artifact from Recipe 223 attached to each release and you have an evidence trail: what ran, where, against which version, and what the gate saw when it opened.

Chapter Summary

- CI/CD transforms tests from manual to automatic
- Smoke tests on every PR (fast feedback, 2-5 minutes)
- Full regression on merge/nightly (comprehensive, all browsers)
- Upload screenshots as artifacts for debugging CI failures
- Use matrix strategies for parallel multi-browser testing
- Generate HTML reports for team visibility
- Deployment gates prevent broken code from reaching production
- Playwright Docker images include all browser dependencies
- Pin the Playwright image tag to the Playwright version in package.json, upgrade them together
- Stamp reports with version, environment, and browser metadata so runs stay comparable across projects

Next: Testing CMS Applications, patterns specific to content management systems.

Chapter 26: Testing CMS Applications

Testing logic and recipes by Rajab Natshah, drawn from the Varbase acceptance suite.

Content Management Systems (WordPress, Drupal, Joomla, headless CMS platforms) have unique testing patterns. Content workflows involve creating drafts, editing, publishing, unpublishing, and managing media. User roles in a CMS are more granular than in a typical app, content creators, editors, reviewers, publishers, and administrators each have different permissions.

CMS testing is also where the one-team-many-projects reality bites hardest. This is the world I come from: at Vardot we maintained the Varbase distribution, and one suite pattern had to hold across dozens of Drupal sites that shared the same editorial workflows but each carried its own theme, modules, and content model. The recipes in this chapter are shaped by that experience, they lean on labels and roles instead of markup, because labels and roles are what survive a CMS upgrade.

This chapter provides recipes tailored to CMS testing patterns.

Recipe 225 Testing Content Creation Workflow

Problem: A CMS earns its keep the moment an editor writes something, which makes content creation the flow you can least afford to break. The moment it hurts is the Tuesday you roll out a core or plugin update: a JavaScript regression in the editor quietly breaks “Save as Draft”, and by lunchtime every editor on the client’s team is blocked, and their support ticket reaches you before your own monitoring does. You want the editor’s first job of the day, write an article and save a draft, proven on every build instead of discovered by the client.

Solution:

```
Feature: Content Creation
  As a content editor
  I want to create articles
  So that I can publish content on the website

  Scenario: Create a new article
    Given I am a logged in user with the "editor" user
    When I go to "/admin/content/add"
    And wait for AJAX to finish
    And I fill in "Title" with "Getting Started with BDD"
    And I fill in "Body" with "Behaviour-Driven Development is a
practice..."
    And I select "Technology" from "Category"
    And I press "Save as Draft"
    And wait for AJAX to finish
    Then I should see "Content saved"
    And I should see "Status: Draft"

  Scenario: Create article with rich text
    Given I am a logged in user with the "editor" user
    When I go to "/admin/content/add"
    And wait for AJAX to finish
    And I fill in "Title" with "Formatted Article"
    And I fill in the rich text editor "Body" with "Bold and italic
content"
    And I press "Save as Draft"
    And wait for AJAX to finish
    Then I should see "Content saved"
```

How It Works: `Given I am a logged in user with the "editor" user` performs a real login as the named test user, so every step after it runs with editor permissions, the same lens the real editor sees through. The fill steps do more than they look like: `I fill in "Title" with ...` resolves the field label-first, falling back to id, name, and placeholder only when no label matches. That ordering is the survival trait for CMS tests, because admin themes love to rewrite ids and class names between versions while the human-visible label “Title” stays put. The rich-text step exists because WYSIWYG editors (CKEditor, TinyMCE) replace the textarea with their own

editing surface, typing into the hidden textarea would do nothing, so the dedicated step targets the editor itself. The `wait for AJAX to finish` lines hold the scenario while the admin page does its constant background chatter, autosave, editor initialization, autocomplete lookups, by watching `window.__webshipAjaxCount`, the fetch/XHR counter webship-js injects into every page, until in-flight requests reach zero. And after every state-changing step, webship-js runs its smartSettle check automatically (requests at zero, no pending timers, DOM quiet for 250ms), so the explicit waits are belt and braces around the heaviest moments, not the only thing holding back a race.

At scale: If your team runs one platform across many sites, one Drupal distribution on thirty client projects, say, this feature can be literally identical everywhere, with only `LAUNCH_URL` changing per site. That is how the Varbase suite worked: one set of editorial features, one environment variable per site, and a fleet-wide run after every release candidate. The discipline that makes it possible: keep role names and test-user fixtures identical across projects ("editor" means the same thing on every site), and never assert on theme-specific markup.

Recipe 226 Testing Content Publishing

Problem: Everyone tests their own step of the publishing workflow and nobody tests the handoffs. The editor checks that drafts save, the admin checks that the publish button works, and then a permissions tweak breaks "Submit for Review", content piles up in the pending queue for two days, and the marketing lead asks why the product launch post never went live. Workflow bugs live between roles, so the test has to cross roles too.

Solution:

```
Feature: Content Publishing Workflow
  Scenario: Draft → Review → Publish
    # Step 1: Editor creates draft
    Given I am a logged in user with the "editor" user
    When I go to "/admin/content/add"
    And I fill in "Title" with "New Product Launch"
    And I fill in "Body" with "We are launching a new product..."
    And I press "Save as Draft"
    And wait for AJAX to finish
```

```
    Then I should see "Status: Draft"

    # Step 2: Editor submits for review
    When I press "Submit for Review"
    And wait for AJAX to finish
    Then I should see "Status: Pending Review"

    # Step 3: Switch to admin for approval
    When I click "Logout"
    And wait for AJAX to finish
    Given I am a logged in user with the "admin" user
    When I go to "/admin/content"
    And wait for AJAX to finish
    Then I should see "Pending Review" in the "New Product Launch" row

    # Step 4: Admin publishes
    When I click "Edit" in the "New Product Launch" row
    And wait for AJAX to finish
    And I press "Publish"
    And wait for AJAX to finish
    Then I should see "Status: Published"

    # Step 5: Verify on public site
    When I go to "/articles/new-product-launch"
    And wait for AJAX to finish
    Then I should see "New Product Launch"
    And I should see "We are launching a new product"

  Scenario: Unpublish content
    Given I am a logged in user with the "admin" user
    When I go to "/admin/content"
    And wait for AJAX to finish
    And I click "Edit" in the "New Product Launch" row
    And wait for AJAX to finish
    And I press "Unpublish"
    And wait for AJAX to finish
    Then I should see "Status: Draft"
```

How It Works: The first scenario walks the full lifecycle, draft, review, publish, public verification, switching identity twice along the way. Each `Given I am a logged in user with the ... user` mid-scenario re-authenticates as a different role, which mirrors production exactly: different humans, different permissions, one piece of content. The status assertions after every transition (`Status: Draft`, `Status: Pending Review`, `Status: Published`) pin the CMS's state machine, so a workflow module update cannot silently skip or rename a state without going red. The final step is the one teams forget: publishing is not done when the admin sees "Published", it is done when the anonymous visitor sees the article, which is why the scenario ends on the public URL. And yes, this looks long for the "one behaviour per scenario" rule. It still honors it: the behaviour under test is the lifecycle itself. Splitting it into five scenarios would test five buttons and lose the handoffs, which is where the bugs are.

Pro Tip: The "Unpublish content" scenario assumes "New Product Launch" already exists, which means it depends on the previous scenario having run. Fine in a linear demo, fragile under `--parallel` where scenario order is not guaranteed. Make it stand alone by seeding the published article first, through the API steps if your CMS exposes a content API, or by repeating the publish steps, so every scenario owns its own data.

At scale: The publishing workflow is the contract between your team and every editorial team you serve. Tag these scenarios `@workflow @critical` and run them across the whole fleet after every CMS core or security update, when a distribution update goes out to thirty sites, this is the suite that tells you which two sites have a customized moderation flow that just broke, before their editors find out for you.

Recipe 227 Testing Media Library

Problem: Media upload is the most plugin-entangled flow in a CMS, image styles, file validators, CDN hooks, and editor integrations all pile onto it. So it breaks in interesting ways: a plugin update changes the upload endpoint, and on Friday afternoon marketing is staring at a modal spinner that never stops while the hero image for Monday's campaign sits on their desktop. Upload and insert-into-content are two separate promises; test both.

Solution:

```
Feature: Media Library
  Scenario: Upload an image to the media library
    Given I am a logged in user with the "editor" user
    When I go to "/admin/media"
    And wait for AJAX to finish
    And I press "Upload"
    And wait for the modal to appear
    And I attach the file "test-image.png" to "File Upload"
    And wait for AJAX to finish
    Then I should see "test-image.png"

  Scenario: Insert image into article
    Given I am a logged in user with the "editor" user
    When I go to "/admin/content/add"
    And I fill in "Title" with "Article with Image"
    And I press "Insert Media"
    And wait for the modal to appear
    And I click "test-image.png" in the modal
    And I click "Insert" in the modal
    And wait for the modal to disappear
    And I press "Save as Draft"
    And wait for AJAX to finish
    Then I should see "Content saved"
```

How It Works: `I attach the file "test-image.png" to "File Upload"` sets the file directly on the file input, no operating-system file dialog ever opens, which is what makes this work headless in CI. Keep the fixture image inside the test repo so the path resolves on every machine and every runner. The upload itself is an XHR, so `wait for AJAX to finish` holds the scenario until the request count drains back to zero, that is the spinner moment, covered. The modal steps wait on the dialog's actual visibility transitions rather than a fixed delay, which matters because media modals load their grid contents asynchronously. The second scenario tests the round trip editors actually care about: open library, pick the image, insert it, save, proving the editor integration, not just the storage.

Pro Tip: Use a tiny fixture image, a one-kilobyte PNG is plenty. Upload throughput is not the behaviour under test, and a 3MB fixture taxes every run on every project while bloating the repo for nothing.

Recipe 228 Testing CMS User Roles

Problem: The client's security review asks "can authors see other people's drafts?" and the honest answer is that nobody knows without clicking around production as four different users. Role permissions are configuration, configuration drifts, and the drift is invisible: a permissions change meant for one role quietly grants Settings access to editors, and it sits there until an audit, or an incident, finds it. You want the permission matrix executable, so drift shows up as a red scenario the day it happens.

Solution:

```
Feature: CMS Role Permissions
  Scenario: Admin has full access
    Given I am a logged in user with the "admin" user
    When I go to "/admin"
    And wait for AJAX to finish
    Then I should see "Content"
    And I should see "Users"
    And I should see "Settings"
    And I should see "Structure"

  Scenario: Editor can manage content but not users
    Given I am a logged in user with the "editor" user
    When I go to "/admin"
    And wait for AJAX to finish
    Then I should see "Content"
    And I should not see "Users"
    And I should not see "Settings"

  Scenario: Author can only manage own content
    Given I am a logged in user with the "author" user
    When I go to "/admin/content"
    And wait for AJAX to finish
    Then I should see "My Articles"
    And I should not see "All Articles"

  Scenario: Reviewer can approve but not create
    Given I am a logged in user with the "reviewer" user
    When I go to "/admin/content"
    And wait for AJAX to finish
    Then I should see "Pending Review"
    And I should not see "Add New"
```

How It Works: Four scenarios, four roles, a permission matrix in executable form. Each scenario starts a fresh authenticated session as its role and then asserts in both directions, and the negative assertions are the security half of the test. `I should see "Content"` passes trivially on an over-permissioned site; it is `I should not see "Users"` that catches privilege escalation. Note what this actually tests: the rendered admin UI, meaning what the CMS enforces in front of a real user,

not what the role configuration claims in a settings export. That is the rule “test behaviour, not implementation” doing security work. Belt and braces says also verify that direct URLs like `/admin/users` deny access, because hiding a menu item is not the same as denying the route.

Pro Tip: Four logins per run adds up across a fleet. webship-js ships auth-state steps that save and restore the browser’s storage state, log in once per role, save the session, restore it in later scenarios instead of re-driving the login form every time. Keep one scenario per role on the real login path so the login itself stays covered.

At scale: This feature is the cheapest security regression you will ever own. Keep one canonical roles feature, keep the role-to-test-user mapping in per-project configuration rather than hardcoded in steps, and run it across every site after any permission-affecting update. A fleet-wide red on “Editor can manage content but not users” the morning after an update is a security finding caught by a feature file, which is a very good morning, considering the alternative.

Recipe 229 Testing Taxonomy and Categories

Problem: Taxonomy is the connective tissue of a CMS, categories drive menus, listing pages, and feeds, so taxonomy breaks at a distance. Someone renames or deletes a category and three landing pages quietly go empty; free tagging lets “javascript”, “Javascript”, and “java-script” coexist until the tag cloud is a mess nobody can clean without an audit. Creating terms and attaching them to content are the two operations everything else depends on.

Solution:

```
Feature: Taxonomy Management
  Scenario: Create a new category
    Given I am a logged in user with the "admin" user
    When I go to "/admin/categories"
    And wait for AJAX to finish
    And I press "Add Category"
    And I fill in "Name" with "Technology"
    And I fill in "Description" with "Tech-related articles"
    And I press "Save"
    And wait for AJAX to finish
    Then I should see "Technology"

  Scenario: Assign tags to an article
    Given I am a logged in user with the "editor" user
    When I go to "/admin/content/add"
    And I fill in "Title" with "Tagged Article"
    And I fill in "Tags" with "javascript, testing, bdd"
    And I press "Save as Draft"
    And wait for AJAX to finish
    Then I should see "Content saved"
```

How It Works: The first scenario is straight CRUD on the vocabulary: create the term, save, and assert it appears in the listing, persistence proven through the same UI an admin would use. The second covers the editor side. `I fill in "Tags" with "javascript, testing, bdd"` types a comma-separated list into the tags field, and on most CMSs that field is an autocomplete widget firing a lookup request per term, background XHR traffic that the closing `wait for AJAX to finish`, plus the automatic settle webship-js runs after each step, keeps from racing the save. If your taxonomy field creates new terms on the fly, this scenario doubles as a test of that path: save, then check the terms actually landed in the vocabulary.

Real-World Pattern: Editorial Workflow Across Roles

Problem: The features above test one role at a time. Real editorial work is a relay: an author drafts, a reviewer approves, a publisher ships, and the bug you most fear is the one in the *handoff*, where a draft is publicly reachable before anyone approved it. You want a single scenario that walks the content through its whole moderation lifecycle, switching identity at each stage, and proves the gates hold.

Solution:

```
Feature: Editorial Moderation Workflow

  Scenario: A draft moves Draft -> Review -> Published across three roles
    Given I am a logged in user with the "author" user
    When I go to "/node/add/article"
    And I wait for the text "Create Article" to appear
    And I fill in "Title" with "Quarterly Update"
    And I fill in "#edit-body-0-value" with "Draft body copy." by attr
    And I press "Save"
    And I wait for the text "Quarterly Update" to appear
    Then I should see "Draft"

  Scenario: An anonymous visitor cannot reach the unpublished draft
    Given I am an anonymous user
    When I go to "/articles/quarterly-update"
    Then I should see "Access denied"
    And I should not see "Draft body copy."

  Scenario: A reviewer moves it to Published
    Given I am a logged in user with the "reviewer" user
    When I go to "/admin/content/moderated"
    And I wait for the text "Quarterly Update" to appear
    And I click the "Quarterly Update" link
    And I select "Published" from "Change to"
    And I press "Apply"
    And wait for AJAX to finish
    Then I should see "The moderation state has been updated"
```

How It Works: Each scenario starts a fresh authenticated session as a different role, and the order tells the story: create as author, prove the gate as anonymous, promote as reviewer. The middle scenario is the one that earns its keep, `I should see "Access denied"` plus `I should not see "Draft body copy."` proves the unpublished content is sealed both at the route and in the rendered output, the exact privacy promise a moderation workflow exists to keep. The `I am an anonymous user` step clears cookies and starts clean, so it genuinely tests the public view, not a stale editor session. Fill-by-attribute (`#edit-body-0-value ... by attr`) targets the rich-text field directly when its label is not a usable handle.

At scale: This three-role relay is the template large distributions lean on, the Varbase acceptance suite, for instance, structures whole feature groups around “can role X reach state Y?” Keep one moderation feature per content type, map the role names to per-project test users in configuration, and the same scenario protects the editorial gate on every site you run it against. A fleet-wide red on the anonymous-access scenario is a privacy incident caught before release, which is the cheapest place to catch one.

Chapter Summary

- CMS applications have unique content workflow patterns (draft, review, publish, unpublish)
- Test the full publishing lifecycle with multi-user scenarios
- Media library testing covers upload, preview, and insertion into content
- CMS role permissions are more granular than typical apps
- Taxonomy and category management needs CRUD testing
- WYSIWYG/rich text editor testing uses specialized steps
- Content moderation workflows involve multiple user roles
- Negative assertions (I should not see) are the security half of role testing
- One canonical feature set plus per-site LAUNCH_URL scales the same suite across a fleet

Next: Testing Single-Page Applications, handling client-side routing and state.

Chapter 27: Testing Single-Page Applications

Single-Page Applications (React, Angular, Vue.js, Svelte) load one HTML page and then dynamically update content using JavaScript. The URL changes but the page never fully reloads. This creates unique testing challenges: client-side routing means `page.goto()` might not trigger a full page load, state management means data persists between views, and lazy loading means components appear asynchronously.

The good news: webship-js handles SPAs well because Playwright understands client-side navigation. Better still, the smart-wait machinery you met earlier (request counters, timer tracking, DOM-quiet detection) was built with exactly this kind of app in mind. This chapter shows you the patterns that matter.

SPA vs Traditional Page Load

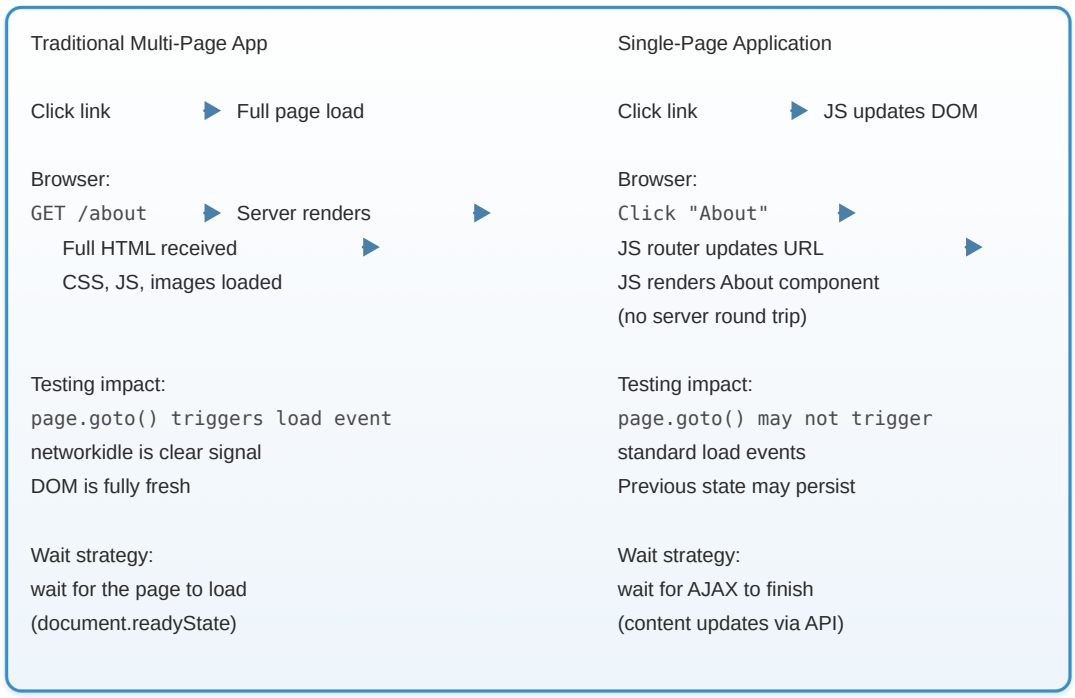


FIGURE 27.1 *SPA vs Traditional Page Load*

Recipe 230 Testing SPA Navigation

Problem: The product was rewritten as a React SPA and the old navigation tests still pass, which should worry you, because they only ever assert on the first page. Meanwhile support tickets say "I clicked back and lost my place", and nobody reproduces it manually because everyone tests forward, never backward. In an SPA, navigation is a JavaScript state change wearing a URL costume; the server is not involved, and a whole class of bugs lives in that difference.

Solution:

```
Feature: SPA Navigation
  Scenario: Navigate between SPA routes
    Given I am on homepage
    Then I should see "Welcome"

    When I click "Products"
    And wait for AJAX to finish
    Then I should see "Product Catalog"

    When I click "About"
    And wait for AJAX to finish
    Then I should see "About Us"

    # Browser back should work with client-side routing
    When I move backward one page
    And wait for AJAX to finish
    Then I should see "Product Catalog"

  Scenario: Direct URL navigation in SPA
    Given I am on homepage
    When I go to "/products"
    And wait for AJAX to finish
    Then I should see "Product Catalog"

    When I go to "/products/laptop-pro"
    And wait for AJAX to finish
    Then I should see "Laptop Pro"
```

How It Works: When the test clicks “Products”, the router calls `history.pushState`, the URL changes, the DOM is rewritten in place, and no request for `/products` ever reaches the server. Playwright clicks the real link like a user would, and webship-js’s settle machinery takes over: a click is a state-changing step, so the framework automatically runs its smartSettle check afterward, watching the in-page counters it installed before any app code ran, `window.__webshipAjaxCount` for in-flight fetch/XHR, `window.__webshipPendingTimers` for queued timeouts, `window.__webshipLastMutation` for DOM churn, and moving on only once requests are at zero, timers are drained, and the DOM has been quiet for 250ms. The explicit

`wait for AJAX to finish` adds a firm checkpoint on the data-heavy transitions. `I move backward one page` drives the browser's real back button (`history.back()`), and the assertion after it catches the classic SPA bug where the router updates the URL but not the view. The second scenario is different in kind: `I go to "/products/laptop-pro"` is a full page load, which tests your hosting's deep-link story, the server must serve the app shell for any route. A 404 there is a hosting-config bug, not an app bug, and this is the only scenario shape that will ever find it.

Key patterns for SPAs:

- Always use `wait for AJAX to finish` after navigation (data loads via API)
- Test browser back/forward (common SPA bug: client-side routing breaks history)
- Test direct URL navigation (deep linking must work)
- Test page refresh (SPA must handle server-side rendering or redirect)

At scale: Across a portfolio of SPAs, React on one project, Vue on the next, this exact feature runs unchanged, because no step ever names the framework. Make it the first feature every new SPA project gets, tagged `@navigation @smoke`. The deep-link and back-button scenarios earn their keep at fleet level: hosting configs differ per project even when the app pattern is identical, and these two scenario shapes are what catch that class of per-project drift.

Recipe 231 Handling Client-Side Routing

Problem: A test flakes right after navigation, so the new hire does the intuitive thing on their first PR and adds `wait for the page to load`. It passes on their laptop, the dev server is slow enough to mask the race, and fails in CI, where everything resolves instantly. The confusing part: the wait is not waiting. In an SPA the load event fired once, minutes ago, on the first page; every later "page" is a DOM update, and a step that watches load state has nothing left to watch.

Solution:

```
# In a SPA, use these patterns:

# After clicking a navigation link:
When I click "Products"
And wait for AJAX to finish     ← Waits for API data to load
Then I should see "Products"

# After form submission:
When I press "Save"
And wait for AJAX to finish     ← Waits for API response
Then I should see "Saved"

# For direct URL navigation:
When I go to "/dashboard"
And wait for AJAX to finish     ← API calls made on route change
Then I should see "Dashboard"

# NOT this (only works for full page loads):
# And wait for the page to load    ← May resolve immediately in SPA
```

How It Works: `wait for the page to load` watches document readiness, `document.readyState` and the load event. Those signals fire on real navigations, and in an SPA there is exactly one of those: the first. Every later route change resolves the step immediately, which is why it "passes" while the view is still half-rendered. `wait for AJAX to finish` waits on actual work instead: before any application script runs, webship-js injects wrappers around `fetch` and `XMLHttpRequest` (via Playwright's `addInitScript`), so `window.__webshipAjaxCount` tracks every in-flight request, and the step holds until that count reaches zero. A route change that fetches data keeps the count above zero for exactly as long as it matters. And here is the part that surprises people: most of the time you need neither step. Every navigation gets a settle budget of up to 3000ms, every state-changing step is followed by an automatic settle check, requests at zero, timers at zero, DOM quiet for at least 250ms, polled every 50ms, and web-first assertions like `Then "sel" should be visible` poll on their own for up to 5 seconds. Keep the explicit AJAX wait for the heavy transitions where you want a named, visible checkpoint in the scenario; a reader should be able to see where the app breathes.

Recipe 232 Testing State Persistence

Problem: A router refactor ships, and the cart badge now resets to zero whenever the user visits the About page, the provider component got unmounted on route change, and nobody noticed because unit tests check the cart logic, not the cart's survival across navigation. State in an SPA lives in JavaScript memory, which means every route change is an opportunity to lose it. The Tuesday before release is a bad time to learn which refactor took that opportunity.

Solution:

```
Feature: SPA State Persistence
  Scenario: Cart persists across navigation
    Given I am on homepage
    When I go to "/products"
    And wait for AJAX to finish
    And I press "Add to Cart"
    Then I should see "1 item in cart"

    # Navigate to another page
    When I click "About"
    And wait for AJAX to finish
    Then I should see "About Us"
    # Cart count should still show
    And I should see "1 item in cart"

    # Navigate back to products
    When I click "Products"
    And wait for AJAX to finish
    And I should see "1 item in cart"

  Scenario: Form data survives route change (if designed to)
    Given I am a logged in user with the "admin" user
    When I go to "/settings"
    And wait for AJAX to finish
    And I fill in "Display Name" with "Rajab"
    # Navigate away without saving
    When I click "Dashboard"
    And wait for AJAX to finish
    # Navigate back
    When I click "Settings"
    And wait for AJAX to finish
    # Behavior depends on app design: data may or may not persist
```

How It Works: Client-side state (Redux, Pinia, context, signals, the pattern is the same) survives route changes only because the page never reloads, and these scenarios assert that survival directly by re-checking “1 item in cart” after every transition. Each click is followed by webship-js’s automatic settle, requests drained, timers empty, DOM quiet, so the badge assertion runs after the re-

render finishes, not during it. The second scenario is deliberately open-ended, and the comment is the point: whether unsaved form data survives navigation is a product decision, not a testing one. Keep the scenario as a placeholder until the product owner decides, then turn the comment into assertions, a feature file is exactly the right place for that decision to leave a record everyone can read. One distinction worth keeping sharp: surviving a route change is memory; surviving a reload is storage. If the cart must outlive a refresh, test that separately with `When I reload the page` and assert the persistence layer, webship-js's cookie steps (`Then a cookie with the name "cart" should exist`) cover the cookie case.

At scale: State-persistence bugs are dependency-upgrade bugs, they arrive when a project bumps its router or state library, and across fifteen SPAs that is happening somewhere every month. Tag these scenarios `@state` and wire them into dependency-update PRs (Renovate, Dependabot), because this is precisely the class of regression that unit tests miss and changelogs never mention.

Recipe 233 Testing Lazy-Loaded Components

Problem: The dashboard test passes every time on office wifi and fails one run in five on CI, where a loaded runner takes 1.8 seconds to deliver the chart bundle. Code splitting means parts of your app arrive on demand, the route renders, then the chunk downloads, then the component mounts, then its data loads, and a test that assumes "page open means page complete" is betting against the network four times per view.

Solution:

```
Feature: Lazy Loading
  Scenario: Dashboard widgets load progressively
    Given I am a logged in user with the "admin" user
    When I go to "/dashboard"
    And wait for AJAX to finish
    # First widget loads quickly
    Then I should see "Quick Stats"

    # Charts may load later (lazy component + API data)
    And wait for AJAX to finish
    Then I should see "Revenue Chart"
    And I should see "Traffic Chart"

  Scenario: Route-based code splitting
    Given I am on homepage
    # Admin module is lazy-loaded
    When I go to "/admin"
    And wait for AJAX to finish
    # Web-first assertion polls until the lazy chunk renders
    Then "h1" should contain text "Admin Dashboard" within 15 seconds
```

How It Works: The first scenario reads as repetitive, two AJAX waits, and that repetition is the mechanics of lazy loading: the first wait drains the route's initial data requests, then the chart components mount and fire their own fetches, and the second wait drains those. Two waves of work, two checkpoints. The second scenario is where the old reflex reaches for `wait 1s`, and webship-js 2.0 takes a hard line here: there are no static sleeps anywhere in the framework, and your features should match. A sleep is wrong twice at once, too long for the fast case (a one-second tax on every green run, forever) and too short for the slow one (the loaded CI runner takes 1.8 seconds and you flake). There is also a subtle reason the settle machinery alone cannot promise this one: lazy chunks arrive via dynamic `import()`, which goes through the browser's module loader, not through `fetch` or XHR, so the request counter can sit at zero, timers drained, DOM quiet, while the chunk is still on the wire. The web-first assertion bridges exactly that gap: `should contain`

`text` polls the DOM every 100ms and succeeds the instant the heading renders. A fast machine pays a couple hundred milliseconds; a slow one gets the full 15-second budget; nobody pays for time they do not need.

Pro Tip: If a chunk regularly needs more than the default 5-second assertion window, that is a performance finding to hand your developers, not a timeout to keep raising. The `within N seconds` clause is for documented exceptions, not a pressure-relief valve.

Recipe 234 Testing API-Driven Components

Problem: The user-list test fails with "John Smith not found" and the failure screenshot shows a table full of skeleton loaders, the component mounted, the assertion ran, and the API response arrived a beat later. Components that fetch on mount put a network round trip between "the page is there" and "the data is there", and every assertion you write is racing it.

Solution:

```
Feature: API-Driven Components
  Scenario: User list loads from API
    Given I am a logged in user with the "admin" user
    When I go to "/admin/users"
    And wait for AJAX to finish
    Then I should see "John Smith"
    And I should see "Jane Doe"

  Scenario: Infinite scroll loads more data
    Given I am on homepage
    When I go to "/feed"
    And wait for AJAX to finish
    Then I should see "Post 1"
    When I scroll to bottom
    And wait for AJAX to finish
    Then I should see "Post 11"
```

How It Works: The mount-fetch-render sequence is the heartbeat of an API-driven SPA, and `wait for AJAX to finish` is aimed straight at it: the step holds until `window.__webshipAjaxCount`, the counter maintained by the fetch/XHR wrappers webship-js injects into every page, drains to zero, meaning the mount-time requests have all resolved. The infinite-scroll scenario extends the same idea. `I scroll to bottom` brings the sentinel element into view, the intersection observer fires, the next page request goes out, the counter rises, and the same wait catches it on the way back down. Asserting "Post 11" then proves page two actually rendered, not merely that a request happened. As a second line of defense, web-first assertion steps poll for up to 5 seconds on their own, so a response that lands a beat after the AJAX wait clears still gets caught before a false failure.

At scale: The hidden coupling in this recipe is the data: "John Smith" has to exist in every environment you point the suite at, and across many projects that quietly becomes a fixture-management job nobody owns. Two clean ways out: seed what the scenario needs through the API steps in a `Background`, or make the test hermetic by mocking the endpoint with `Given the URL "**/api/users" returns the JSON:` so the UI is tested against data the test controls. Keep a thin `@smoke` layer that hits real data per environment, and let everything else own its inputs.

Testing Different SPA Frameworks

Framework-Specific Considerations

React:
- Components re-render on state changes
- useEffect hooks trigger API calls after render
- wait for AJAX catches async useEffect data fetching

Angular:
- Change detection may delay DOM updates
- Observables resolve asynchronously
- wait for AJAX handles HttpClient requests

Vue.js:
- nextTick needed for DOM updates after data changes
- Vue Router transitions may need short waits
- `wait for AJAX handles axios/fetch calls`

Universal Pattern (works for all):

```
When I click "Link"
And wait for AJAX to finish          ◀ handles async data
Then I should see "Content"          ◀ verifies DOM updated
```

webship-js does not need framework-specific configuration.
Playwright sees the final DOM regardless of framework.

FIGURE 27.2 *Testing Different SPA Frameworks*

Chapter Summary

- SPAs update the DOM via JavaScript, not full page reloads
- Use `wait for AJAX to finish` instead of `wait for the page to load` for SPAs
- Test browser history (back/forward), a common SPA bug
- Test direct URL navigation (deep linking)
- Test state persistence across route changes
- Lazy-loaded chunks arrive outside the request counters, use web-first assertions, never static sleeps
- webship-js works with React, Angular, Vue, and any SPA framework
- Playwright sees the final DOM regardless of which framework rendered it

Next: Building a Complete Test Suite, a worked example from start to finish.

Chapter 28: Building a Complete Test Suite

This chapter brings everything together. We will build a complete test suite for a realistic e-commerce application, from project setup to CI configuration. Every recipe in this book is a building block. Now we assemble them.

One framing before we start: we are not just writing tests, we are building an asset the whole team operates. The measure of a complete suite is not coverage, it is whether a new hire can clone it, run it, and read last night's report on their first morning, and whether the same structure can be stamped onto the next project without a redesign.

The Application: ShopDemo

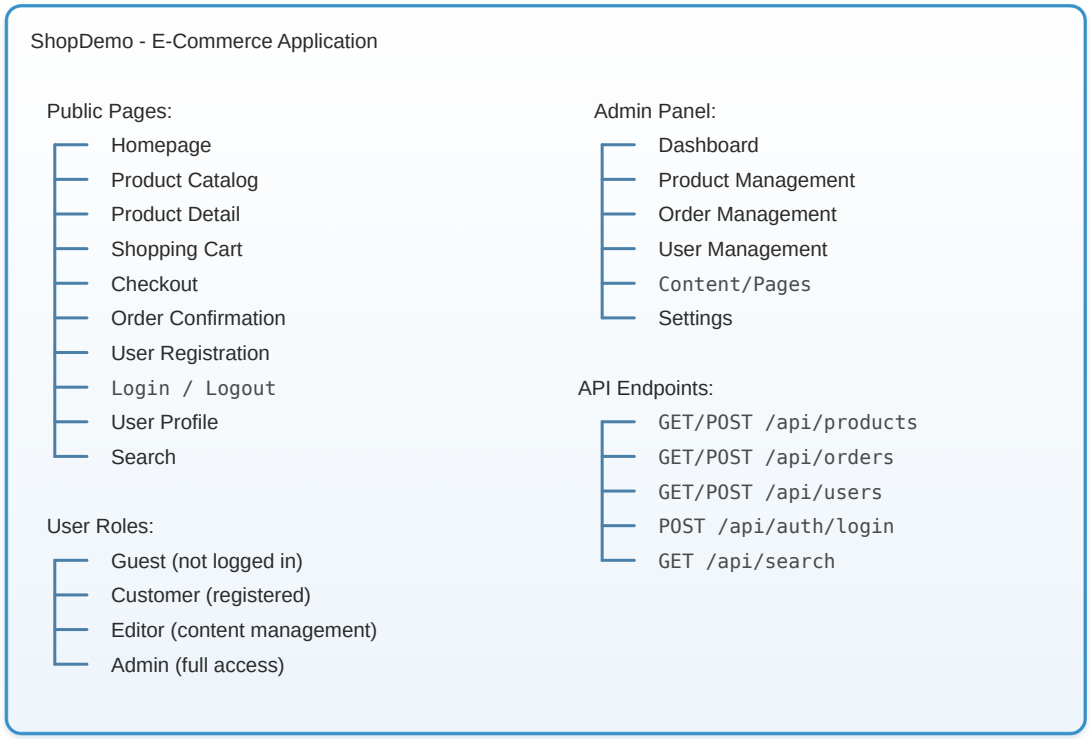


FIGURE 28.1 *The Application: ShopDemo*

ShopDemo is deliberately shaped to exercise every pattern in this book: form-heavy auth flows, a stateful cart, admin CRUD behind role-gated routes, a JSON API, and public pages that have to clear an accessibility bar. Four roles give us a permission matrix; the API gives us a fast non-browser tier. If your real application is a CMS or a SaaS dashboard instead of a shop, the shapes still map one-to-one, swap “product” for whatever your domain calls a record.

Complete Project Structure

```
shopdemo-tests/
├── package.json
├── cucumber.js                          ◀ Configuration profiles
├── playwright.config.ts                 ◀ Browser config
│
├── tests/
│   ├── features/
│   │   ├── auth/
│   │   │   ├── login.feature
│   │   │   ├── registration.feature
│   │   │   └── logout.feature
│   │   ├── products/
│   │   │   ├── catalog.feature
│   │   │   ├── search.feature
│   │   │   └── product-detail.feature
│   │   ├── cart/
│   │   │   ├── add-to-cart.feature
│   │   │   └── cart-management.feature
│   │   ├── checkout/
│   │   │   └── checkout-flow.feature
│   │   ├── admin/
│   │   │   ├── dashboard.feature
│   │   │   ├── product-admin.feature
│   │   │   ├── order-admin.feature
│   │   │   └── user-admin.feature
│   │   ├── api/
│   │   │   ├── products-api.feature
│   │   │   ├── orders-api.feature
│   │   │   └── auth-api.feature
│   │   └── accessibility/
│   │       └── a11y-audit.feature
│   │
│   ├── step-definitions/
│   │   ├── auth-steps.js
│   │   ├── cart-steps.js
│   │   ├── viewport-steps.js
│   │   ├── screenshot-steps.js
│   │   └── a11y-steps.js
│   │
│   ├── support/
│   │   ├── hooks.js
│   │   ├── helpers.js
│   │   └── builders.js
│   │
│   └── assets/
│       └── test-image.png
│
├── screenshots/                         ◀ Auto-generated
├── reports/                             ◀ Auto-generated
└── .github/workflows/test.yml           ◀ CI configuration
```

FIGURE 28.2 *Complete Project Structure*

The layout works with webship-js's defaults instead of against them: Cucumber discovers step definitions from `tests/step-definitions/**/*.js` and features from `tests/features/**/*.feature`, so the structure is convention, not configuration. Features are foldered by business domain, auth, cart, checkout, never by test type, because when checkout breaks you want every checkout scenario in one place, whether it drives a browser or the API. Custom step files stay thin; webship-js's 411 built-in steps carry most scenarios, and `support/builders.js` is where test-data construction lives, which will matter shortly. When selector complexity grows, add a `tests/selectors/` folder with named-selector JSON files so features speak in names and markup churn gets absorbed in one file instead of fifty.

At scale: This tree is your template repository. New project: clone it, change `LAUNCH_URL`, delete the features that do not apply, and the new suite inherits the structure, the CI workflow, and the conventions on day one. Twenty projects with the same skeleton means any QA engineer can navigate any suite. The alternative, twenty hand-grown layouts, is how teams end up with suites only their original author can maintain.

The Feature Files

Authentication

Login goes first and gets the `@smoke` tag, because it is the cheapest possible canary: if login is down, nothing else matters and half the suite cannot even start.

```
# tests/features/auth/login.feature
@smoke @auth
Feature: User Login
  As a registered user
  I want to log in
  So that I can access my account

  Scenario: Successful login
```

```
    Given I am on homepage
    When I go to "/login"
    And I fill in "Email" with "customer@example.com"
    And I fill in "Password" with "CustPass123"
    And I press "Sign In"
    And wait for AJAX to finish
    Then I should see "My Account"
    And I should not see "Sign In"

  Scenario: Failed login
    Given I am on homepage
    When I go to "/login"
    And I fill in "Email" with "customer@example.com"
    And I fill in "Password" with "WrongPassword"
    And I press "Sign In"
    And wait for AJAX to finish
    Then I should see "Invalid credentials"
    And I should see "Sign In"

  Scenario: Empty credentials
    Given I am on homepage
    When I go to "/login"
    And I press "Sign In"
    Then I should see "Email is required"
```

Look at what the steps are quietly doing for you. `I fill in "Email"` resolves the field label-first, it finds the label with that text and follows it to the input, falling back to id, name, and placeholder, so the scenario survives the markup refactor that renames `#edit-email` to `#login-email`. After `I press "Sign In"`, webship-js's settle hook waits for the login request to drain and the DOM to go quiet before the next step fires, and the explicit `wait for AJAX to finish` adds a checkpoint a reader can see. The three scenarios are the three things that actually happen at a login form: right password, wrong password, empty submit. And the negative assertion in the first scenario matters, `I should not see "Sign In"` proves the page genuinely changed state, which the positive assertion alone cannot.

Pro Tip: This feature tests login. The rest of the suite just needs to *be* logged in, which is a different requirement. webship-js ships auth-state steps that save, restore, and clear the browser's storage state: drive the form once, save the state, and let every later scenario restore it instead of retyping credentials (Chapter 10 covers the pattern). With thirty scenarios needing a customer session, that is minutes off every single run.

Shopping Cart

```
# tests/features/cart/add-to-cart.feature
@smoke @cart
Feature: Shopping Cart
  As a customer
  I want to add products to my cart
  So that I can purchase them

  Scenario: Add a product to cart
    Given I am on homepage
    When I go to "/products"
    And wait for AJAX to finish
    And I click "Laptop Pro"
    And wait for AJAX to finish
    Then I should see "Laptop Pro"
    And I should see "$999.99"
    When I press "Add to Cart"
    And wait for AJAX to finish
    Then I should see "Added to cart"

  Scenario: View cart with items
    Given I am a logged in user with the "customer" user
    When I go to "/products"
    And wait for AJAX to finish
    And I click "Laptop Pro"
    And wait for AJAX to finish
    And I press "Add to Cart"
    And wait for AJAX to finish
    When I go to "/cart"
    And wait for AJAX to finish
    Then I should see "Laptop Pro"
```

```
    And I should see "$999.99"
    And I should see "Quantity"

  Scenario: Remove item from cart
    Given I am a logged in user with the "customer" user
    When I go to "/cart"
    And wait for AJAX to finish
    And I click "Remove" in the "Laptop Pro" row
    And wait for AJAX to finish
    Then I should not see "Laptop Pro"
    And I should see "Your cart is empty"
```

The cart is where state starts to bite. The first two scenarios build their own state, they add the laptop before asserting on it, but the third does not: it goes straight to `/cart` and expects "Laptop Pro" to already be there, which is only true because the previous scenario left it behind for the same logged-in customer. That coupling works for months and then explodes the day you add `--parallel`, when scenarios stop running in file order. In a real suite, give the removal scenario its own setup, repeat the add steps, or seed the cart through the API tier, so every scenario owns its state. Meanwhile the row-scoped click is doing quiet, important work: with three products in the cart there are three "Remove" links, and `in the "Laptop Pro" row` is what keeps the step from grabbing the first one it finds.

At scale: Shared test accounts are the multi-suite version of the same trap. When four projects' nightly runs all log in as `customer@example.com` against one shared staging backend, one suite's leftover cart fails another suite's empty-cart assertion, and the report blames the wrong project. Give each suite, and ideally each parallel worker, its own user, and write the account-naming convention into your shared template so nobody rediscovers the collision at 2 a.m.

Admin Product Management

```
# tests/features/admin/product-admin.feature
@regression @admin
Feature: Product Management
  As an admin
  I want to manage the product catalog
```

```
So that customers see accurate product information

Background:
  Given I am a logged in user with the "admin" user
  And I go to "/admin/products"
  And wait for AJAX to finish

Scenario: View product list
  Then I should see "Products"
  And I should see "Laptop Pro"
  And I should see "Wireless Mouse"

Scenario: Create a new product
  When I press "Add Product"
  And wait for AJAX to finish
  And I fill in "Name" with "USB-C Hub"
  And I fill in "Price" with "49.99"
  And I fill in "SKU" with "HUB-001"
  And I select "Accessories" from "Category"
  And I press "Save"
  And wait for AJAX to finish
  Then I should see "Product created"
  When I go to "/admin/products"
  And wait for AJAX to finish
  Then I should see "USB-C Hub"

Scenario: Edit a product
  When I click "Edit" in the "Laptop Pro" row
  And wait for AJAX to finish
  And I fill in "Price" with "899.99"
  And I press "Save"
  And wait for AJAX to finish
  Then I should see "Product updated"

Scenario: Delete a product
  When I click "Delete" in the "USB-C Hub" row
  And wait for the modal to appear
  And I click "Confirm" in the modal
```

```
And wait for AJAX to finish
Then I should see "Product deleted"
And I should not see "USB-C Hub"
```

`Background` runs before every scenario in the feature, log in, navigate, settle, which saves three lines per scenario and, more importantly, makes every scenario start from the same known place. Read the CRUD lifecycle in order, though: create the hub, edit the laptop, delete the hub. That delete depends on the create two scenarios earlier, the same ordering coupling the cart feature has, tolerable in a demo and worth fixing in production (create-and-delete inside one scenario, or seed through the API tier). The modal steps earn their keep on the destructive action: `wait for the modal to appear` polls for the dialog instead of assuming it opened instantly, and `I click "Confirm" in the modal` scopes the click so it cannot hit some other “Confirm” elsewhere on the page.

API Tests

```
# tests/features/api/products-api.feature
@regression @api
Feature: Products API
  Background:
    Given the API base URL is "http://localhost:8080"
    Given I am authenticating as "admin" with "AdminPass123" password

  Scenario: List all products
    When I send a GET request to "/api/products"
    Then the API response code should be 200
    And the response should be valid JSON
    And the API response should contain "Laptop Pro"

  Scenario: Create a product via API
    When I send a POST request to "/api/products" with body:
      """
      {
        "name": "Test Product",
        "price": 19.99,
        "sku": "TEST-001",
        "category": "Test"
      }
      """
    Then the API response code should be 201
    And the JSON response should have "name" equal to "Test Product"
    And the JSON response should have property "id"

  Scenario: Get a single product
    When I send a GET request to "/api/products/1"
    Then the API response code should be 200
    And the JSON response should have property "name"
    And the JSON response should have property "price"

  Scenario: Unauthenticated API access fails
    When I send a GET request to "/api/admin/users"
    Then the API response code should be 200
    # Admin endpoints require admin auth, which we have
```

These scenarios never launch a browser. The API steps drive HTTP requests directly through axios, which is why this whole feature finishes in seconds and makes a perfect early pipeline stage: if `POST /api/products` is broken, you want to know before spending browser minutes watching the admin form fail for the same reason. The `Background` applies to every scenario in the feature, and that is exactly the trap in the last one. Its name says unauthenticated access fails, but the Background has already authenticated as admin, so 200 is all it can ever see; a genuine unauthenticated check has to live in a feature file without that Background. Scenario names are documentation. When the name and the assertion disagree, the next reader inherits the confusion, treat that mismatch as a bug in the test, even when the run is green.

Accessibility

```
# tests/features/accessibility/a11y-audit.feature
@a11y
Feature: Accessibility
  Scenario: Homepage is accessible
    Given I am on homepage
    And wait for the page to load
    Then the page should have no critical accessibility violations

  Scenario: Login page is accessible
    Given I am on homepage
    When I go to "/login"
    And wait for the page to load
    Then the page should have no critical accessibility violations

  Scenario: Product listing is accessible
    Given I am on homepage
    When I go to "/products"
    And wait for AJAX to finish
    And wait for the page to load
    Then the page should have no critical accessibility violations
```

The audit step runs the bundled axe-core engine against the live, rendered page, computed styles, injected widgets and all, and fails only on findings axe rates critical. That is the right opening bar for an application that has never been audited; once the criticals are gone, the stricter sibling `Then`

`the page should pass an accessibility audit at level "AA"` applies the full WCAG AA rule set, and `Then print accessibility violations` dumps each finding with its impact and help URL into the report when you need the detail. Keep the `@a11y` tag on these scenarios: audits cost more time than functional checks and belong in the nightly tier, not in every pull-request run.

The Configuration

```
// cucumber.js
const common = {
  require: [
    'node_modules/webship-js/tests/step-definitions/**/*.js',
    'tests/step-definitions/**/*.js',
    'tests/support/**/*.js',
  ],
  format: ['@cucumber/pretty-formatter'],
};

module.exports = {
  default: {
    ...common,
    worldParameters: {
      launchUrl: 'http://localhost:8080',
      minWaitTime: {
        before_scenario: 0, after_scenario: 0,
        before_step: 0, after_step: 0, page: 3000,
      },
    },
  },
  ci: {
    ...common,
    format: ['@cucumber/pretty-formatter', 'json:reports/report.json'],
    worldParameters: {
      launchUrl: process.env.LAUNCH_URL || 'http://localhost:8080',
      minWaitTime: {
        before_scenario: 0, after_scenario: 0,
        before_step: 0, after_step: 0, page: 3000,
      },
    },
  },
};
```

Two profiles, one shared `common` block. The `require` array is what wires webship-js in: the first path loads the framework's 411 built-in steps out of `node_modules`, the next two load your custom steps and hooks, delete that first line and every built-in step suddenly "does not exist". `minWaitTime.page: 3000` is the per-navigation settle budget: after each page load, webship-js

gives the application up to three seconds to finish its AJAX, drain its timers, and stop mutating the DOM before the next step runs. The `ci` profile changes exactly two things: it adds a JSON formatter, the input that `bin/generate-reports.js` later turns into HTML and PDF reports, and it reads `LAUNCH_URL` from the environment instead of hard-coding localhost. That second change is the important one.

At scale: `LAUNCH_URL` as an environment variable is the difference between one suite that tests every environment and four diverging copies that each test one. The pipeline sets `LAUNCH_URL=https://staging.example.com` for the staging job and the production URL for the post-deploy smoke job, same repo, same features, zero edits. Apply the rule everywhere: base URLs and secrets live in CI variables, never in committed config. Across twenty projects, that single convention is what lets a platform engineer rewire an environment without opening twenty repos.

Running the Suite

```
# Smoke tests (fast feedback)
npx cucumber-js --tags "@smoke"

# Full regression
npx cucumber-js --tags "not @wip"

# API tests only (no browser needed)
npx cucumber-js --tags "@api"

# Accessibility audit
npx cucumber-js --tags "@a11y"

# Full suite, all browsers, parallel
BROWSER=chromium npx cucumber-js --parallel 4 --profile ci
```

The tag tiers map straight onto pipeline stages: `@smoke` on every push for feedback in minutes, the full regression before a deploy, `@api` as the cheap early gate that needs no browser at all, `@a11y` on the nightly schedule. `--parallel 4` starts four Cucumber workers, each driving its

own browser, which roughly quarters the wall time and doubles as an honesty test: any scenario leaning on another scenario's leftovers (remember the cart) starts failing intermittently the day you switch it on. Better to find that out now than during release week.

At scale: The JSON report is the half of this output your managers actually consume, once `bin/generate-reports.js` turns it into HTML with metadata stamped in: `--app-version 2.3.1 --env staging --browser chromium`. Use identical metadata keys in every project's pipeline, because that is what makes a cross-project dashboard possible. And that dashboard is the real answer to "is release X safe?", a green run with a version number on it that a delivery manager can read unaided, instead of a QA engineer's recollection relayed in a status meeting.

Chapter Summary

- A complete test suite covers: auth, CRUD, cart, checkout, API, admin, accessibility
- Organize by business domain, not by test type
- Use tags for tiered execution: @smoke, @regression, @api, @a11y
- Background steps reduce repetition within a feature
- API tests complement browser tests (faster, more precise)
- CI configuration ties everything together

Next: BDD in the Age of AI, how artificial intelligence transforms testing.

Chapter 29: BDD in the Age of AI

Concepts and field notes by Shaimaa Natshah, from the Webship.co blog.

Everything you have learned in this book is about to become ten times more powerful. AI does not replace testing, it amplifies it. The developers who master BDD and automated testing will not be the ones replaced by AI. They will be the ones who direct AI to build higher-quality software, faster.

This is my vision for the future of software development, and it starts with a simple framework: **Test, Drive, Develop**.

The AI-Development Revolution

We are living through the biggest shift in software development since the invention of high-level programming languages. AI can now:

- Generate code from natural language descriptions
- Write unit tests for existing code
- Suggest bug fixes from error messages
- Refactor code for readability and performance

But AI has a fundamental limitation: **it does not know what "correct" means for your business.** It can write code that compiles, runs, and looks reasonable, but without tests, there is no way to verify that the code does what the business actually needs.

This is where BDD becomes the most valuable skill in the AI age.

The Value Equation in the AI Age

Code generation	Commoditized (AI does it)
Test writing	Amplified (AI assists, humans guide)
Business understanding	Premium (only humans have this)
Quality assurance	Critical (AI output needs validation)

The person who writes the tests controls the quality.
The person who controls the quality controls the product.

FIGURE 29.1 *The AI-Development Revolution*

Run that equation across a portfolio and it gets sharper. AI multiplies the amount of code a team can produce, which means it multiplies the amount of unverified code a team can produce. One team shipping twenty projects with AI assistance does not need twenty times the reviewers. It needs executable specifications on every project, so that “does it still work?” is a question a pipeline answers, not a person.

The Test-Drive-Develop Framework

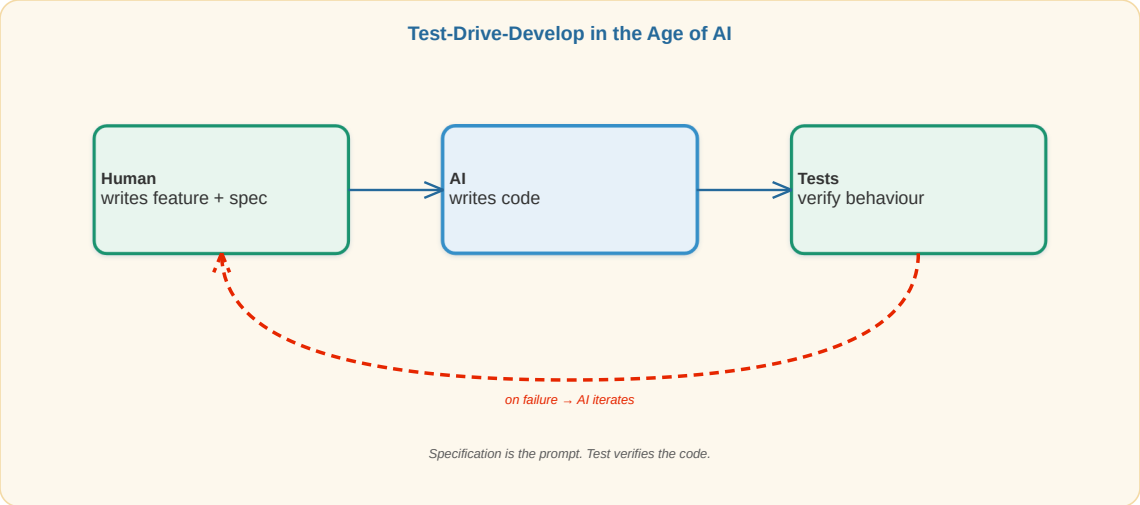


FIGURE 29.2 *The Test-Drive-Develop Framework*

Traditional TDD says: write a test, make it pass, refactor. I propose an evolution for the AI age:

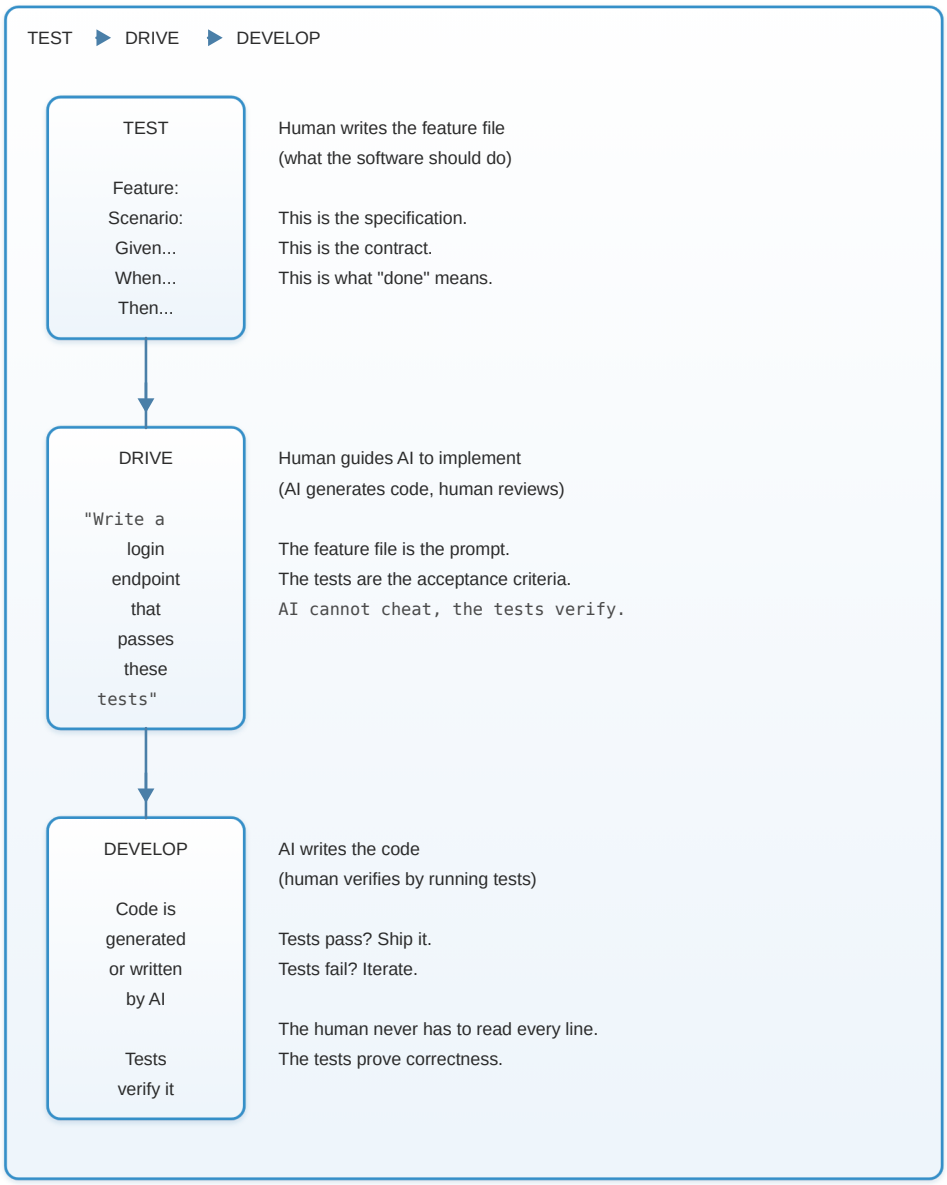


FIGURE 29.3 *The Test-Drive-Develop Framework*

Notice what this framework changes about your job. You stop being the person who types the code and become the person who defines “done”, and “done” is written in Gherkin that runs. The feature file is simultaneously the prompt, the acceptance criteria, and the regression suite. Write it once and it keeps earning: first as the AI’s instructions, then as the proof the AI delivered, then as the guard that next month’s AI session does not quietly break this month’s work.

Recipe 235 Using AI to Generate Feature Files

Problem: A registration module lands in the sprint, and the acceptance criteria live in a ticket, three bullet points and a screenshot. You need scenarios ready for Thursday’s review session, and the engineer who would normally write them is already covering two other projects. This is exactly the moment to let AI draft the Gherkin, and exactly the moment most teams get burned by merging the draft unread.

Approach:

Give AI the business requirements and ask it to generate Gherkin:

```
Prompt to AI:
"I have a user registration feature. Users need to provide
name, email, and password. Password must be at least 8 characters.
Email must be unique. On success, redirect to dashboard.
Generate BDD feature files using webship-js Gherkin style."
```

AI generates:

```
Feature: User Registration
  As a new user
  I want to create an account
  So that I can access the application

  Scenario: Successful registration
    Given I am on homepage
    When I go to "/register"
    And I fill in "Name" with "Jane Doe"
    And I fill in "Email" with "jane@example.com"
    And I fill in "Password" with "SecurePass123"
    And I fill in "Confirm Password" with "SecurePass123"
    And I press "Create Account"
    And wait for AJAX to finish
    Then I should see "Dashboard"

  Scenario: Password too short
    Given I am on homepage
    When I go to "/register"
    And I fill in "Name" with "Jane Doe"
    And I fill in "Email" with "jane@example.com"
    And I fill in "Password" with "short"
    And I fill in "Confirm Password" with "short"
    And I press "Create Account"
    Then I should see "Password must be at least 8 characters"

  Scenario: Duplicate email
    Given I am on homepage
    When I go to "/register"
    And I fill in "Name" with "Jane Doe"
    And I fill in "Email" with "admin@example.com"
    And I fill in "Password" with "SecurePass123"
    And I fill in "Confirm Password" with "SecurePass123"
    And I press "Create Account"
    And wait for AJAX to finish
    Then I should see "Email already registered"
```

Human review checklist:

- Do the scenarios match the business requirements?
- Are edge cases covered?
- Is the language correct for our application's UI labels?
- Are the step patterns compatible with our webship-js setup?

How It Works: The prompt does two jobs. The business half ("password must be at least 8 characters, email must be unique") gives the model rules to turn into scenarios, including the unhappy paths a rushed human skips. The "webship-js Gherkin style" half anchors the output to step phrasings that actually exist: `I fill in "Email" with ...` resolves the field by its visible label first, then falls back to id, name, and placeholder; `I press` clicks the named button; `I should see` asserts visible text. Because every line maps to one of the 411 built-in steps, the generated file runs with zero step-definition work, and your review can focus on the one thing AI cannot verify, whether these are the right scenarios for the business. One more detail worth knowing: in webship-js 2.0, the `wait for AJAX to finish` lines are belt and braces. After every state-changing step, an AfterStep hook runs smartSettle, which watches `window.__webshipAjaxCount`, pending timers, and DOM mutations until the page has been quiet for 250ms. The suite stays stable even where the model forgets a wait.

At scale: By the third project you onboard, the prompt itself becomes an asset. Keep a prompt template in the repo with your house rules baked in, UI label conventions, your tag taxonomy, the `page--test-category.feature` naming scheme, so every AI draft across the portfolio starts out consistent instead of drifting with whoever typed the prompt that day. One shared prompt is cheaper to maintain than twenty diverging suite styles.

Recipe 236 Using AI to Generate Step Definitions

Problem: The review session produced a clean feature file, but one line, `When I add {string} to the cart`, matches nothing in the built-in steps. The teammate who knows the World object inside out is on leave, the PR is due tomorrow, and you would rather not reverse-engineer locator chaining from two-year-old code tonight. AI gets you 90% of a custom step in seconds; the skill is knowing which 10% to check.

Approach:

```
Prompt to AI:
"I have this Gherkin step: 'When I add {string} to the cart'
Write a Playwright step definition for a webship-js project.
The page has product cards with 'Add to Cart' buttons."
```

AI generates:

```
When('I add {string} to the cart', async function (productName) {
  const card = this.page.locator('.product-card')
    .filter({ hasText: productName });
  await card.locator('button', { hasText: 'Add to Cart' }).click();
  await this.page.waitForLoadState('networkidle');
});
```

Human review: Does this selector match our actual HTML? Test it.

How It Works: The generated code is genuinely good Playwright. `this.page` is the live Page object webship-js exposes on the World, and `.filter({ hasText: productName })` narrows the locator to the one card containing the product name, which keeps working when the page gains a second row of cards. The line to scrutinize is the last one. webship-js 2.0 already runs `smartSettle(page, 1500)` in an AfterStep hook after every state-changing step, so the explicit `waitForLoadState('networkidle')` is usually redundant, and on pages with long-polling or analytics beacons, `networkidle` may never fire at all, quietly burning your 45-second step timeout. Models reproduce their training data, and the internet is full of pre-smartSettle code. Keep the locator, trim the wait.

At scale: Before accepting any AI-written step, grep your shared step library first. The ugliest portfolio smell is five projects each carrying a slightly different “add to cart” implementation because AI rewrote it five times. Keep custom steps in one shared package where you can, and hold AI output to the same repo rule as human output: every new step documented with at least five example lines.

Recipe 237 AI-Assisted Test Maintenance

Problem: Design ships a rebrand on Monday: “Sign In” becomes “Log In”, “Email” becomes “Username”. Tuesday’s nightly run is a wall of red, 30 scenarios across six features, all dead on the same two strings. Fixing it by hand is thirty minutes of error-prone find-and-replace. Explaining to your delivery manager why the quality dashboard stays red all week is worse.

Approach:

1. Run the test suite, collect all failures
2. Give AI the error messages and the old step text
3. AI suggests updated steps based on the new UI

```
Prompt to AI:
"These tests fail because the 'Sign In' button was renamed to 'Log In'
and the 'Email' field was renamed to 'Username'. Update these feature
files."
```

AI performs a bulk find-and-replace that would take a human 30 minutes in 30 seconds.

How It Works: The failures all carry the same signature, and that is what makes this safe to delegate. Each web-first assertion polled for the old label every 100ms for its full five-second window before timing out, so the report hands you a precise list of which strings died and where. Feed AI that list plus the rename mapping, and the sweep is mechanical. The non-negotiable part is the re-run: AI will happily “fix” a scenario whose real problem was a behavior change hiding behind the rename. A green run after the sweep proves the fix; a human reading the diff proves nothing got papered over.

At scale: Rename storms hit every project that shares a design system. Fix one project, then reuse the exact same prompt on the other nineteen. Better still, this pain is the argument for named selectors: for volatile elements, keep the targeting in `tests/selectors/*.json` files, webship-js auto-loads them through `worldParameters.selectors.files`, so a rebrand becomes a one-line JSON edit per project instead of a 30-file sweep. It is a trade: you give up some plain-language readability for rename resilience. Spend it on navigation chrome and brand buttons; keep visible labels in the business-critical flows your product owner actually reads.

The Pitfalls of AI-Generated Tests

```
AI Testing Pitfalls

  1. OVER-TRUSTING AI OUTPUT
                                  AI can generate tests that look correct but test the
     wrong thing. Always review generated tests against
     business requirements.

  2. TESTING IMPLEMENTATION, NOT BEHAVIOR
     AI tends to test what the code does, not what the
     business needs. Human oversight ensures tests stay
     behavior-focused.

  3. MISSING EDGE CASES
     AI generates the happy path well but may miss
                                 business-critical edge cases that only domain experts
     know about.

  4. FALSE CONFIDENCE
     "We have 500 AI-generated tests!" means nothing if
                                they do not cover the scenarios that actually matter.

  THE RULE: AI generates, humans validate.
```

FIGURE 29.4 *The Pitfalls of AI-Generated Tests*

The fix for all four pitfalls is the same: treat “AI generates, humans validate” as a gate, not a slogan. Put AI-generated scenarios through the same review as human-written ones, one behavior per scenario, wait for events not time, test behavior not implementation. The moment a team starts merging generated tests unread because “the AI knows the framework,” the suite stops being a specification and becomes decoration.

Structured Prompt-Driven Development (SPDD)

> **NOTE**
>
> Coined by Thoughtworks, championed by Martin Fowler. SPDD turns AI prompts into versioned engineering artifacts.

The Test-Drive-Develop loop in this chapter is one half of the picture. The other half is what *kind of prompt* you hand the AI. Vibe coding, “build me a login screen”, works for prototypes. It collapses for production. **Structured Prompt-Driven Development (SPDD)** restores engineering discipline.

The REASONS Canvas

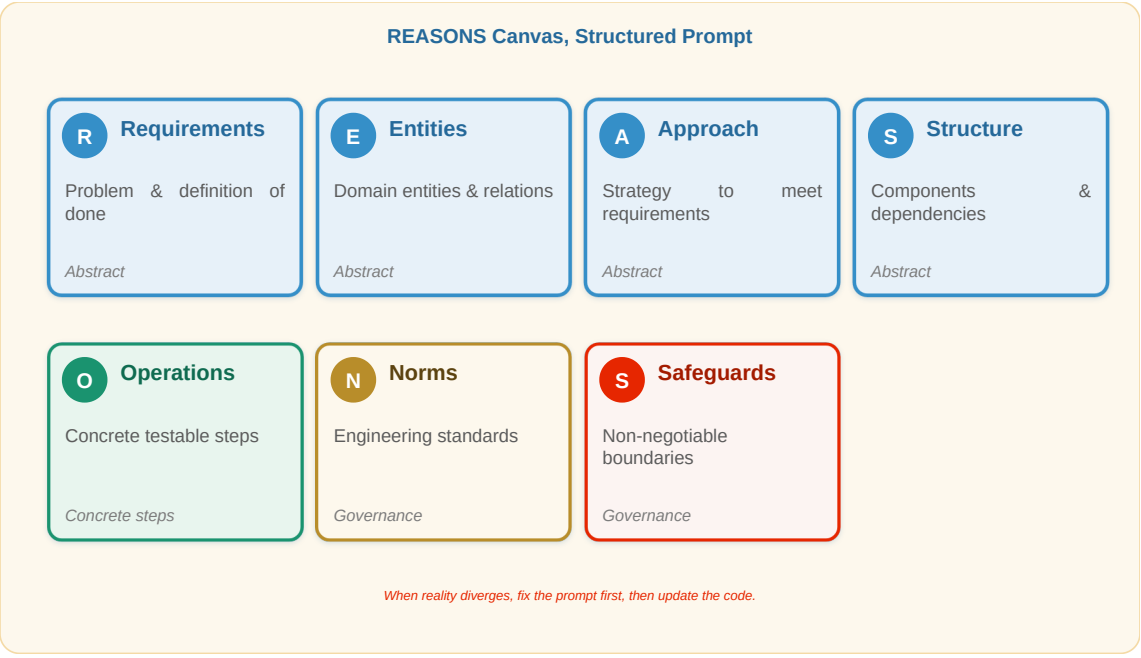


FIGURE 29.5 *The REASONS Canvas*

The seven sections move from abstract intent to concrete execution. Webship-js feature files map directly:

REASONS SECTION	WEBSHIP-JS ARTIFACT
Requirements	`Feature:` description + As/I want/So that
Entities	Domain nouns referenced in step text
Approach	Background section + scenario outlines
Structure	`tests/` layout + `cucumber.js` require:
Operations	Individual `Given / When / Then` steps
Norms	`worldParameters` + locale/a11y scenarios
Safeguards	Tags (`@critical` , `@security`) + assertion scenarios

The Cardinal Rule

> **IMPORTANT**
>
> When reality diverges from the prompt, **fix the prompt first**, then update the code.

This single rule is what separates SPDD from spec-driven development. Specs drift once coding begins. SPDD prevents that decay by treating the prompt as the source of truth and regenerating code from it whenever the spec changes.

Three Required Skills

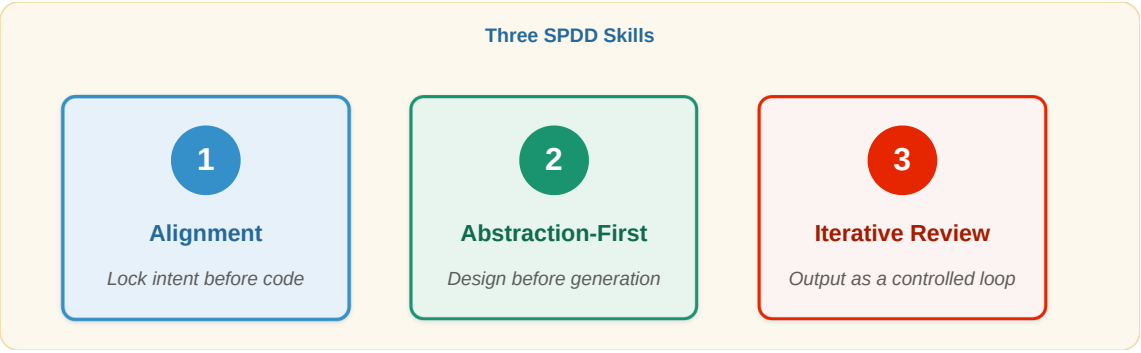


FIGURE 29.6 *Three Required Skills*

1. **Alignment**, lock intent before you write code.
2. **Abstraction-First**, design before you generate.
3. **Iterative Review**, output is a controlled loop, not a one-shot.

Webship-js gives you the verification layer that SPDD demands. The Gherkin file is the structured prompt. The step definitions are the executable verification. Together they form a closed loop: AI generates code -> tests verify -> failures point at the prompt -> prompt or code gets refined -> regenerate.

Chapter 35 covers SPDD with webship-js in depth, including REASONS-canvas templates, AI agent prompts, and CI-level prompt-code sync gates.

The Future of Testing

The developers who thrive in the AI age will not be the fastest coders. They will be the ones who:

1. **Understand the business** well enough to write correct specifications
2. **Write tests first** that define “done” before a single line of code exists
3. **Direct AI** to implement against those tests
4. **Validate** that the implementation actually meets the specification

This is the BDD workflow, supercharged by AI. The test is the specification. The specification is the prompt. The prompt produces the code. The test verifies the code.

The Testing Advantage:

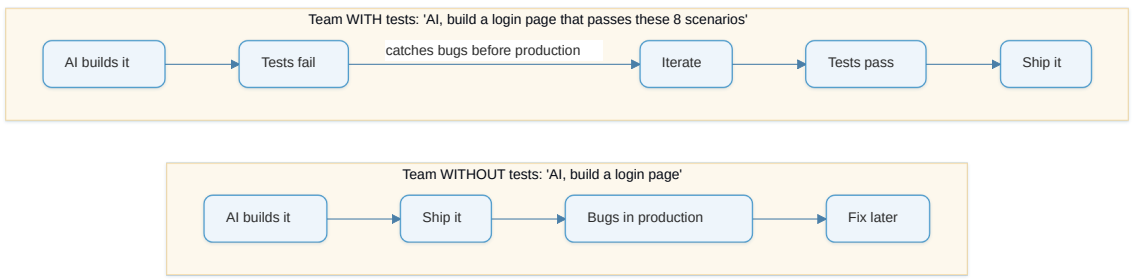


FIGURE 29.7 *The Future of Testing*

Result: 40-80% fewer production bugs (industry research).

Chapter Summary

- AI amplifies testing, it does not replace it
- The Test-Drive-Develop framework: humans write tests, AI writes code, tests verify correctness
- AI can generate feature files, step definitions, and test maintenance updates
- Always review AI-generated tests against business requirements
- The person who writes the tests controls the quality
- BDD skills are MORE valuable in the AI age, not less
- Tests are the specification that AI implements against
- The future belongs to developers who combine business understanding with testing discipline

Next: Best Practices and Pro Tips.

Chapter 30: Best Practices and Pro Tips

Concepts and field notes by Shaimaa Natshah, from the Webship.co blog.

This chapter distills everything so far into actionable rules. After more than two hundred recipes, here are the principles that decide whether a suite is still trusted, and still readable, three years and twenty projects later.

Writing Great Scenarios

Recipe 238 The DAMP Principle

Problem: A scenario named “Checkout works” fails, and the engineer on triage duty opens it to find helper steps calling other helpers, abbreviations only the original author understood, and setup hidden two files away. That author moved to another project six months ago. The new hire’s first PR review takes an hour, not because the change is hard, but because nobody can read the tests it touches.

Solution: DAMP, Descriptive And Meaningful Phrases. Every scenario should be understandable without looking at anything else.

```
# DAMP: Self-contained, clear
Scenario: Customer completes checkout
  Given I am a logged in user with the "customer" user
  When I go to "/cart"
  And wait for AJAX to finish
  Then I should see "Laptop Pro"
  When I press "Checkout"
  And wait for AJAX to finish
  And I fill in "Address" with "123 Main St"
  And I press "Place Order"
  And wait for AJAX to finish
  Then I should see "Order Confirmed"
```

How It Works: DAMP is the deliberate counterweight to DRY. In production code, duplication is the enemy; in test code, opacity is. A scenario gets read far more often than it gets written, usually during a failure, often by someone who did not write it, so a little repetition that keeps the whole story on one screen is a good trade. Every line above can be taken at face value: log in, open the cart, see the product, check out, see the confirmation. When line seven fails, the report names the exact business action that broke, and the explicit `wait for AJAX to finish` lines double as synchronization points anyone can read, no hidden magic to decode.

At scale: DAMP is what lets one QA lead review twenty projects. Self-contained scenarios are portable: a reviewer needs zero knowledge of each project's private helper vocabulary to judge whether a test is right. Make it a portfolio rule, any scenario must read aloud as a sensible user story without opening another file, and reviews stop depending on tribal knowledge.

Recipe 239 One Scenario, One Behavior

Problem: A forty-step scenario called "Complete user journey" goes red in the 2 a.m. CI run. It failed at step 23, but was the bug in login, search, or checkout? Nobody knows without replaying the whole run, and the delivery manager is waiting to hear whether the release is blocked. One scenario, three behaviors, zero diagnosis.

Solution: Each scenario tests exactly one business rule. If a scenario fails, you should know exactly which behavior is broken.

```
# BAD: Tests three behaviors in one scenario
Scenario: Everything
  # Tests login, then product search, then checkout
  # If it fails, which behavior is broken?

# GOOD: One behavior per scenario
Scenario: Successful login
  ...

Scenario: Product search returns results
  ...

Scenario: Checkout processes payment
  ...
```

How It Works: When each scenario owns one behavior, the scenario name IS the diagnosis. "Product search returns results" red, everything else green, triage is finished before you open a single screenshot. The cost is real but small: every scenario starts its own fresh browser context, so three scenarios launch three contexts instead of one. webship-js keeps that overhead low because there are no static sleeps anywhere, smartSettle moves on the moment the page goes quiet, and the seconds spent on extra setup repay themselves the first time you skip a twenty-minute replay.

At scale: Across 40 suites, scenario-name-as-diagnosis is what makes a cross-project dashboard readable. A delivery manager scanning red rows sees broken behaviors in plain language, not step traces. It also keeps your tags honest: `@critical` on a single behavior means something;

`@critical` on a forty-step journey means everything and nothing.

Recipe 240 Scenarios as Stories

Problem: The product owner writes acceptance criteria in a ticket. QA rewrites them as tests. Three weeks later the two have drifted apart, and nobody notices until UAT. Sign-off becomes "trust me, the suite is green", because the one person who knows what the feature should do cannot read the thing that supposedly proves it.

Solution: The best scenarios read like user stories:

```
Scenario: New customer discovers and purchases a product
  Given I am on homepage
  When I fill in "Search" with "wireless keyboard"
  And wait for AJAX to finish
  Then I should see "Wireless Keyboard Pro"
  When I click "Wireless Keyboard Pro"
  And wait for AJAX to finish
  Then I should see "$49.99"
  When I press "Add to Cart"
  And wait for AJAX to finish
  Then I should see "Added to cart"
```

How It Works: Every line is written in the user's vocabulary, search, see the price, add to cart, yet every line executes. `I fill in "Search"` finds the field by its visible label first, exactly the way the user does; `I should see "$49.99"` polls the page's visible text the way a human eyeballs it. There is no translation layer between the acceptance criteria and the test, so there is nothing to drift. A green run of this scenario is evidence a product owner can read without an engineer interpreting, which is what turns sign-off from a trust exercise into a reading exercise. Write acceptance criteria in this shape from day one, and QA stops rewriting them from scratch.

Common Anti-Patterns

Recipe 241 Avoiding God Steps

Problem: Somewhere around the third project, someone writes `Given the system is fully set up with admin user and products and orders`, one step, two hundred lines of JavaScript behind it. Project four needs nine products instead of ten, so the step grows a flag. Project five adds another. Eighteen months later it is four hundred lines, every suite in the portfolio depends on it, and an engineer whose setup is not quite "fully" spends a day debugging a framework instead of a test.

Solution: A "God Step" does too much in a single step definition. Break it into specific, composable steps:

```
# BAD: God step
Given the system is fully set up with admin user and products and orders

# GOOD: Specific, composable steps
Given I am a logged in user with the "admin" user
And I go to "/admin/products"
And wait for AJAX to finish
```

How It Works: A God step hides the test's preconditions inside code where no reader can see them, the Gherkin says "fully set up" and only the implementation knows what that means this month. Composable steps keep the preconditions visible in the scenario itself: who is logged in, which page we are on, what data exists. When something breaks, the failing line names the exact precondition that did not hold. Small steps are also honest about scope: a step that does one thing can serve fifty scenarios without surprising any of them.

At scale: God steps are the number one reason teams cannot share a step library across projects, every project's "fully set up" means something different, so every project forks the step, and now you maintain five versions. Small composable steps share cleanly: login is login everywhere. Keep

framework-level steps in one shared package and let project-specific data setup live in project-specific steps, where it belongs.

Recipe 242 Avoiding Coupled Scenarios

Problem: “Edit the product” has passed for eight months, because “Create a product” happens to run first. Then you enable parallel workers to cut the nightly run from forty minutes to twelve, and it starts failing. Only in CI. Only sometimes. The team burns two days on browser versions and timeouts before anyone notices the test never created the product it edits.

Solution: Scenarios must be independent. Never rely on another scenario’s side effects:

```
# BAD: Scenario 2 depends on Scenario 1
Scenario: Create a product
  ...product is created...

Scenario: Edit the product
  # ASSUMES the product from Scenario 1 exists!
  When I click "Edit" in the "New Product" row

# GOOD: Each scenario creates its own data
Scenario: Edit a product
  Given I am a logged in user with the "admin" user
  And there is a product named "Test Widget"
  When I click "Edit" in the "Test Widget" row
```

How It Works: webship-js already isolates the browser side for you, every scenario starts in a fresh incognito context, so cookies, localStorage, and sessions cannot leak between tests. The coupling that survives lives in your application’s data: rows one scenario creates and another silently assumes. The fix is for each scenario to declare its own world in its Given steps. And when login is the expensive part, that does not mean re-typing credentials in every scenario, webship-js’s auth steps can save the logged-in storage state once and restore it per scenario. Share the session mechanics; never share the test data.

At scale: Ordering bugs surface exactly when you need speed most, parallel workers, sharded CI, selective re-runs of only the failed scenarios. Make independence a review gate with a one-line test: every scenario must pass when run alone, straight from its file and line number. If it cannot, it is not a test. It is a fragment of one.

Recipe 243 Avoiding Implementation Testing

Problem: The team migrates auth from a JWT in localStorage to httpOnly session cookies. The behavior is identical, users never notice. But sixty assertions about token contents go red, and the suite blocks a refactor that changed nothing a user could see. Two sprints of that, and developers learn the most dangerous lesson a suite can teach: red probably does not mean broken.

Solution: Test what the user experiences, not how the code works:

```
# BAD: Testing implementation
Then the database should have a record in the users table
And the session cookie should be set
And the JWT token should contain the user ID

# GOOD: Testing behavior
Then I should see "Dashboard"
And I should see "Welcome, Rajab"
```

How It Works: Behavior assertions survive refactors because they pin down the contract, not the mechanism. `I should see "Dashboard"` polls the page's visible text for up to five seconds, the same check a human makes, indifferent to whether the session lives in a JWT, a cookie, or carrier pigeons. The flip side: when a requirement genuinely lives below the UI, a compliance rule that a session cookie must exist, say, assert it at that level deliberately. webship-js ships twelve cookie matchers (`Then a cookie with the name "session" should exist`) for exactly those cases. The rule is not "never look under the hood." It is "assert at the level where the requirement lives."

At scale: Implementation-coupled suites punish platform upgrades portfolio-wide: upgrade the framework under twenty sites and watch tests fail on markup no user ever sees. Behavior-level suites are how a small team re-validates dozens of sites after every core update, the suites assert what visitors and editors experience, so they keep passing while the internals churn. That is the difference between an upgrade week and an upgrade quarter.

Anti-Pattern Decision Tree

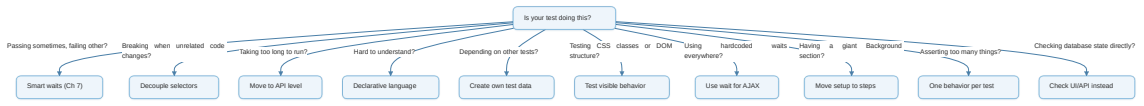


FIGURE 30.1 *Anti-Pattern Decision Tree*

Pin this tree where the team can see it, then go one step further and make it the order of questions in every flaky-test triage and every feature-file review. The answers all live in this book; the tree just stops the team from debating which question to ask first.

Team Practices

Recipe 244 Running Three Amigos Sessions

Problem: The product owner writes “user can log in” in a ticket. The developer builds OAuth with magic links. QA tests email-and-password. Everyone discovers the three different features they built during UAT, the Tuesday before release, the most expensive possible moment to learn that

“done” had three definitions.

Solution: Before writing tests, get three perspectives:

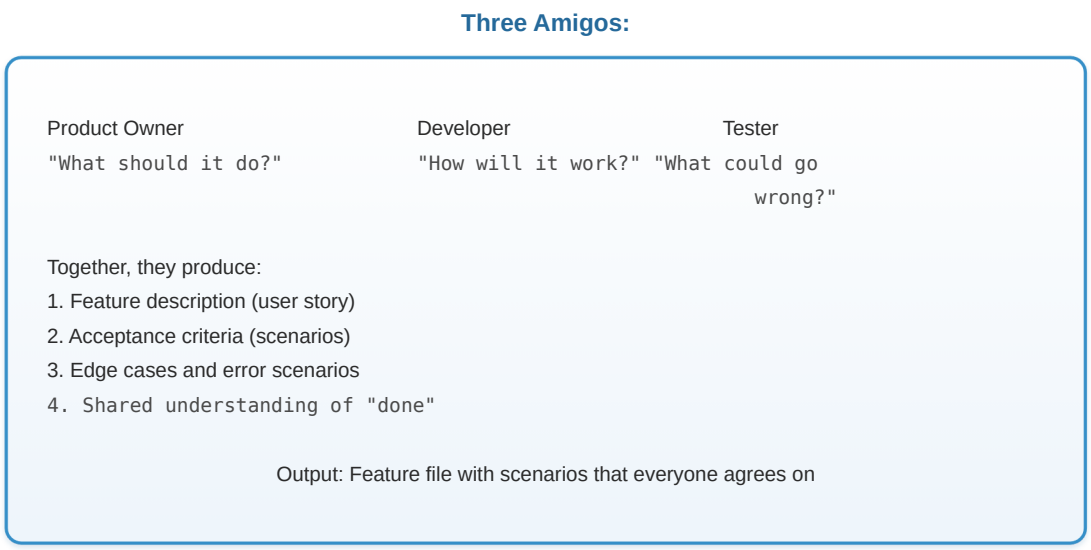


FIGURE 30.2 *Running Three Amigos Sessions (Recipe 244)*

How It Works: The output of a Three Amigos session is not meeting minutes, it is the feature file. The product owner confirms the scenarios describe the right behavior in the right words. The developer confirms each step is automatable before anyone builds anything. The tester adds the failure modes the other two never think of: the duplicate email, the expired session, the double-click on Submit. Thirty minutes per feature, and the scenarios everyone agreed on become the scenarios the pipeline enforces, “done” is now executable. And across many projects, this does not have to be a meeting: have the product owner draft scenarios in the PR and let dev and QA review them as code. The conversation moves into PR comments; the artifact stays exactly the same.

Recipe 245 Code Review for Features

Problem: Feature-file PRs get rubber-stamped because reviewers are not sure what to look for, “it’s just Gherkin.” Three months of LGTM later, the suite is slow, coupled, and flaky, and it got that way one approved PR at a time. The new hire’s first PR is where your conventions get taught, or lost.

Solution: When reviewing a PR with feature files:

```
Feature File Review Checklist:
  ☐ Scenarios describe behavior in business language
  ☐ Each scenario is independent (no shared state)
  ☐ Appropriate tags (@smoke, @regression, @wip)
  ☐ Background is short (3-5 lines max)
  ☐ No hardcoded wait times where smart waits work
  ☐ Assertions check both positive AND negative cases
  ☐ Step text is reusable (not too specific to one page)
  ☐ Feature description explains the "why"
```

How It Works: Each line of the checklist blocks a failure mode from this book. Independence (Recipe 242) gets checked with the run-alone test. Tags are not decoration, they are what CI keys on, so a mistagged scenario silently skips your smoke gate. A long Background is a God step in disguise (Recipe 241). And in webship-js there is rarely a reason to see a hardcoded wait in a diff: smartSettle and the web-first assertions already wait for events, so any `wait N seconds` deserves a review comment asking what it is hiding.

At scale: One checklist across every repo is the cheapest cross-project quality control you will ever ship, it turns reviewer judgment into team convention. Put it in the PR template so it shows up in every review, and rotate reviewers across projects: the fastest way to spread a convention is a reviewer who just saw it done right somewhere else.

The Golden Rules

The 10 Golden Rules of BDD Testing

1. Write tests BEFORE code (or at least before you ship)
2. One scenario = one behavior
3. Each scenario creates its own test data
4. Wait for events, not time
5. Test behavior, not implementation
6. Use business language, not developer jargon
7. Keep scenarios independent
8. Run tests on every push
9. Fix failing tests immediately
10. The test suite is a living document

FIGURE 30.3 *The Golden Rules*

Three of these are hard rules in the webship-js repository itself, one scenario, one behavior; wait for events, not time; test behavior, not implementation. They are not style preferences. They were earned by maintaining the same suite pattern across dozens of sites and watching what survived.

Final Words

You now have everything you need to build a world-class test suite with webship-js, Playwright, and Cucumber-js. The recipes in this book cover every pattern you will encounter: navigation, forms, modals, tables, APIs, authentication, multi-browser testing, accessibility, CI/CD, and more.

But recipes alone do not make a great test suite. What makes a great test suite is the discipline to:

- Write tests first
- Run them on every change
- Fix them when they break
- Keep them readable for the whole team

The test suite is not just a safety net. It is a living specification of your application. It documents what the software does, proves that it works, and catches regressions before they reach users.

In the age of AI, this discipline is not just good practice, it is your competitive advantage. The teams that master BDD will ship faster, with fewer bugs, and with higher confidence than those who rely on manual testing or hope.

Start with one test. Make it pass. Write another. Build the habit. The compound interest of a growing, green test suite is one of the most powerful forces in software development.

The chapters ahead push all of this further, AI agents that write and run webship-js tests for you, real-world walkthroughs of international sites, and the complete step reference. The principles in this chapter are what keep every bit of it honest.

Happy testing.

Part VII: The AI-Powered Testing Ecosystem

Chapter 31: The Webship-js AI Agent Ecosystem

Everything in this book so far has taught you how to write tests. This chapter teaches you how to let AI write them for you, and still be in control.

The webship-js project does not stop at a testing framework. It includes a complete ecosystem of AI tools that plug into Claude Code, Anthropic's AI-powered coding assistant. These tools turn Claude into a testing specialist, one that knows every step definition in webship-js, understands BDD, and can set up, write, run, and debug tests without you touching a single configuration file.

This is the Test-Drive-Develop framework from Chapter 29, made real.

The Three Products

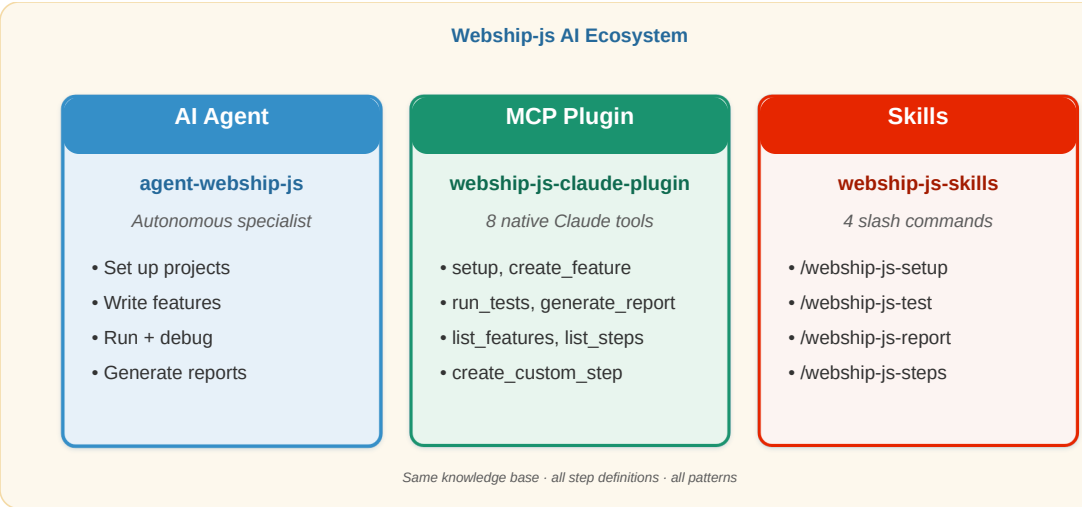


FIGURE 31.1 *The Three Products*

Understanding the Architecture

Before we dive in, let me explain how these three tools relate to Claude Code:

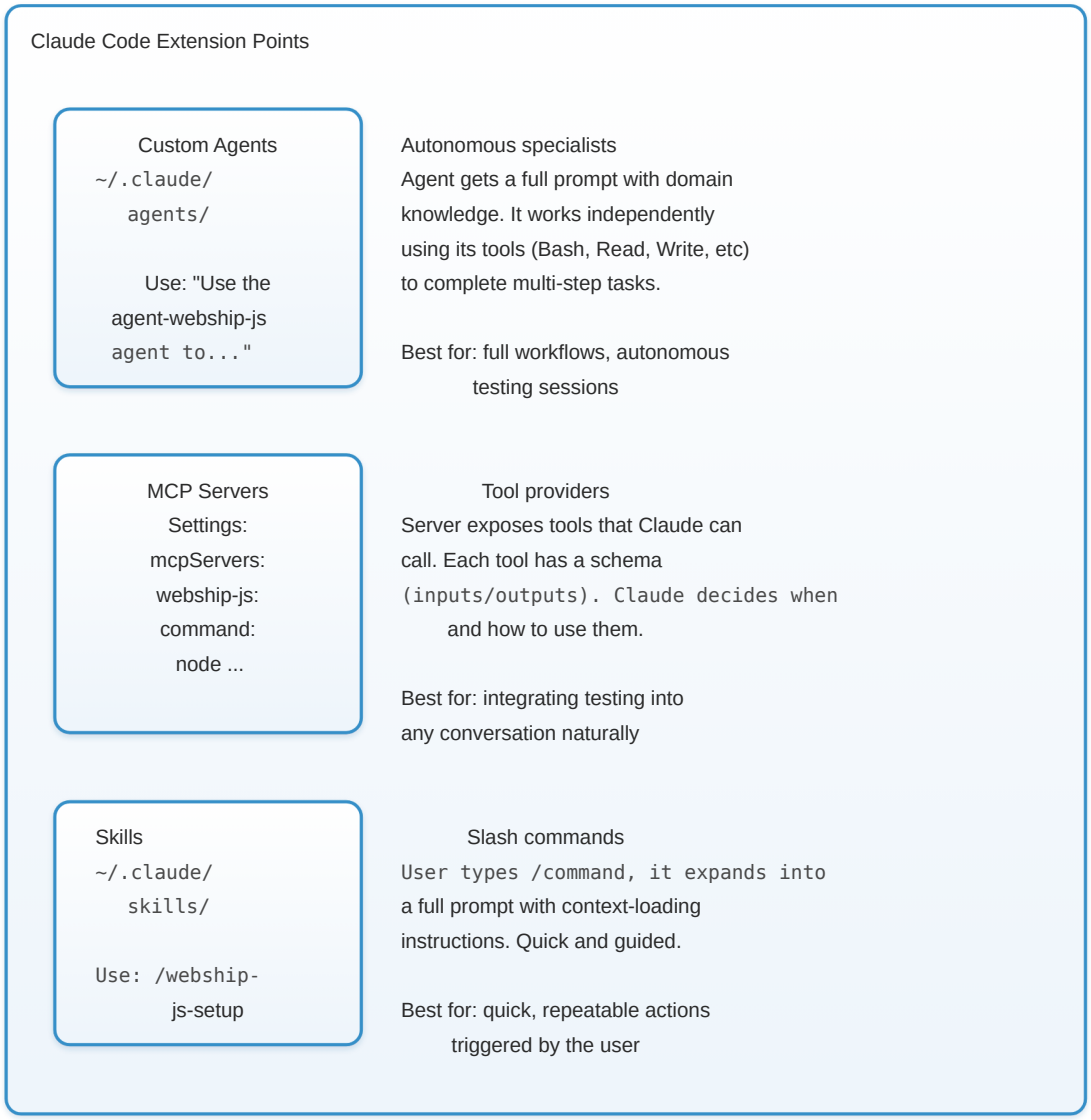


FIGURE 31.2 *Understanding the Architecture*

You do not need all three. Pick the one that fits your workflow:

- **Agent**: “I want Claude to handle everything autonomously”
- **MCP Plugin**: “I want Claude to have testing tools available in any conversation”
- **Skills**: “I want quick slash commands for common testing tasks”

Or use all three together, they complement each other.

Part 1: The Webship-js AI Agent

What It Is

The `agent-webship-js` is a Claude Code custom agent, a markdown file that turns Claude into a specialized testing agent. When invoked, Claude gets the full webship-js knowledge base: every step definition, every configuration option, every debugging trick. It then uses its tools (Bash, Read, Write, Edit, Glob, Grep, WebFetch) to autonomously set up projects, write tests, run them, and fix failures.

Recipe 246 Installing the AI Agent

Problem: Claude Code can already write code in your repo, but ask it to “test the contact page” and it improvises: generic Playwright scripts, invented step phrasings, no idea webship-js exists. You spend the session correcting it instead of reviewing tests. What you want is Claude arriving pre-loaded with the whole webship-js playbook before you type a single prompt.

Solution:

```
# Clone the repository
git clone https://github.com/webship/webship-js-ai-agent.git

# Option 1: Install globally (available in all projects)
cp webship-js-ai-agent/.claude/agents/agent-webship-js.md
~/.claude/agents/

# Option 2: Install to a specific project
cp webship-js-ai-agent/.claude/agents/agent-webship-js.md \
   /path/to/your/project/.claude/agents/

# Or use the install script
cd webship-js-ai-agent
bash install.sh --global
# or
bash install.sh --project /path/to/your/project
```

After installation, the agent file lives at:

- Global: `~/.claude/agents/agent-webship-js.md`
- Project: `<project>/.claude/agents/agent-webship-js.md`

How It Works: There is no binary to install, the agent is a markdown file with frontmatter (name, description, model, tools) followed by a long, carefully engineered prompt. Claude Code discovers agents by scanning `~/.claude/agents/` and the current project's `.claude/agents/` directory; when you invoke one, Claude loads that file as its operating instructions. Global install means every repo on your machine can summon the agent. Project install means the agent travels with the repo.

At scale: For a team, the project install wins. Commit `.claude/agents/agent-webship-js.md` to your project template repo and every engineer, and every new project, gets the same agent version on clone. Upgrading the agent becomes a reviewable PR instead of a "did everyone re-copy the file?" chore across laptops. Treat the agent file exactly like a dependency: versioned, reviewed, rolled out deliberately.

Recipe 247 Using the AI Agent for Full Workflow

Problem: A stakeholder asks for test coverage on a marketing site nobody on your team has touched. The honest estimate is a full day: scaffold the project, explore every page, write the features, debug the failures. You have three other projects this week. You want that day compressed into one sentence you type while reviewing someone else's PR.

Solution: Here is the power move. You tell Claude to use the agent, and it handles everything:

```
You (in Claude Code):
"Use the agent-webship-js agent to test https://myapp.com/contact"
```

The agent then autonomously:

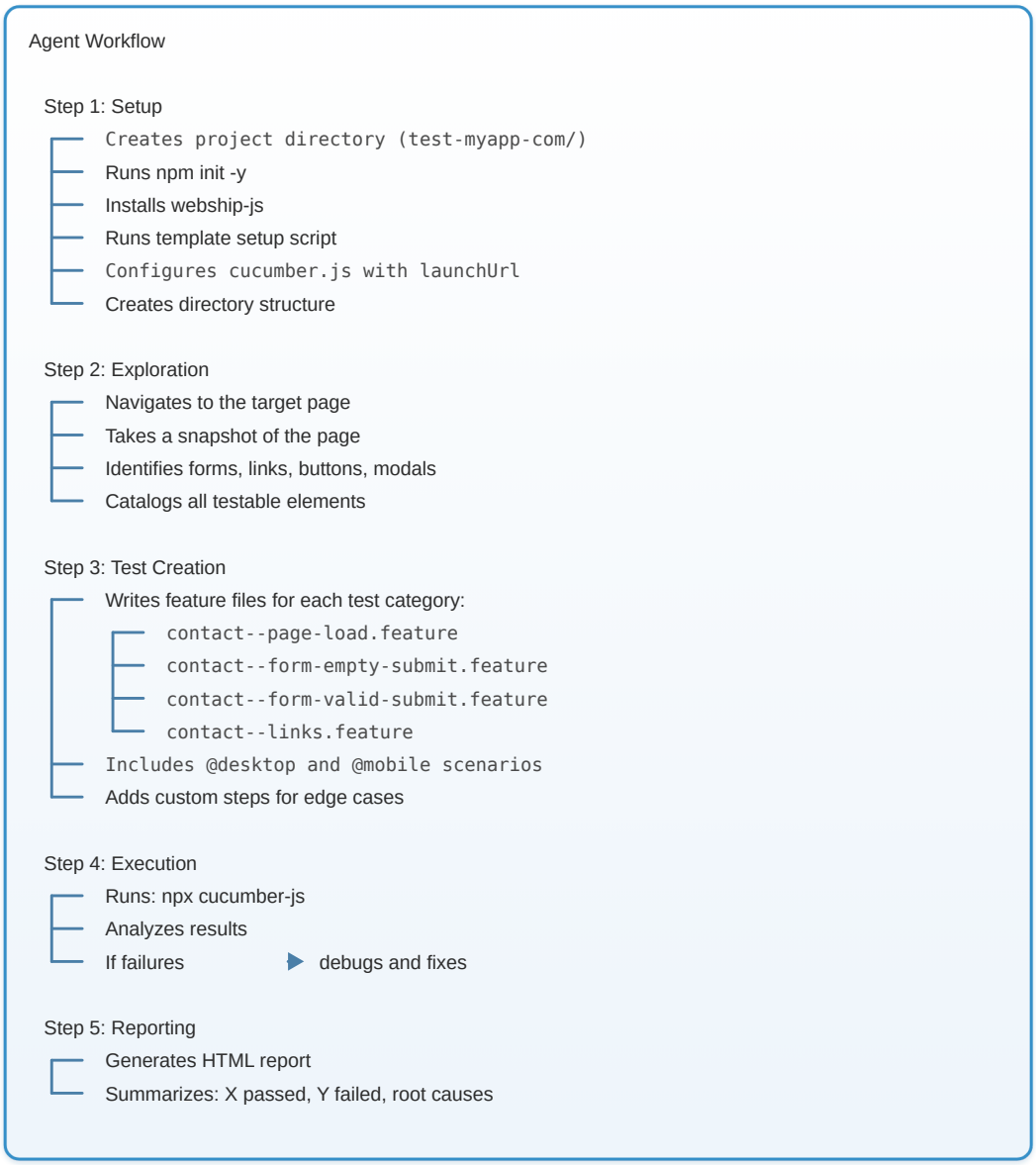


FIGURE 31.3 *Using the AI Agent for Full Workflow (Recipe 247)*

How It Works: The agent runs the same loop a senior test engineer would, scaffold, explore, write, run, debug, report, and every artifact it produces is an ordinary webship-js file you can open, diff, and review. The setup step wires `launchUrl` into the configuration, so the same features later run against staging or production by pointing the base URL elsewhere instead of editing tests. The part that keeps you in control is structural: the agent's output is verified by the tests it writes, and the tests are readable Gherkin. You review the feature files and the run results, not a transcript of everything the AI did.

At scale: Speed is the demo; consistency is the real product. Point the agent at ten microsites and you get ten identically structured suites, same directory layout, same `page--test-category.feature` naming, same tag conventions. That uniformity is what lets one QA lead review ten projects, one CI template build all of them, and one dashboard aggregate the reports without per-project glue.

Recipe 248 Agent-Generated Feature Files

Problem: AI-generated tests have a reputation problem, and it is earned: plausible-looking Gherkin stuffed with steps that almost exist, failing as `undefined` before testing anything. Your team tried it once, spent an afternoon fixing phantom steps, and went back to writing everything by hand. The fix is not better luck with prompts, it is an agent that only emits phrasings shipped in the framework.

Solution: When the agent explores a contact page, it produces feature files like this:

```
# tests/features/contact--page-load.feature
@desktop
Feature: Contact page loads correctly
  As an anonymous user
  I want to visit the contact page
  So that I can find contact information

  @desktop
  Scenario: Contact page loads on desktop
    Given I am on "/contact"
    And I wait for AJAX to finish
    Then I should see "Contact"
    And I should see "Your Name"
    And I should see "Business email"
    And I should see "Your message"

  @mobile
  Scenario: Contact page loads on mobile
    Given I am on "/contact"
    And I wait for AJAX to finish
    Then I should see "Contact"
    And I should see "Your Name"
```

```
# tests/features/contact--form-empty-submit.feature
@desktop @validation
Feature: Contact form validation
  As an anonymous user
  I want to see errors when submitting empty form
  So that I know what fields are required

  Scenario: Empty form shows required field errors
    Given I am on "/contact"
    When I press "Submit"
    And I wait for AJAX to finish
    Then I should see "required"
```

```
# tests/features/contact--form-valid-submit.feature
@desktop @submission
Feature: Contact form submission
  As an anonymous user
  I want to submit the contact form with valid data
  So that my message reaches the team

  Scenario: Submit contact form with all fields
    Given I am on "/contact"
    When I fill in "Your Name" with "Jon Smith"
    And I fill in "Company name" with "Webship.co"
    And I fill in "Business email" with "test@webship.co"
    And I fill in "Phone number" with "+212 (367) 487-7851"
    And I select "11-50 employees" from "Team Size"
    And I fill in "Job title" with "Administrative Assistant II"
    And I fill in "Your message" with "This is just an automated testing"
    And I press "Submit"
    And I wait for AJAX to finish
    Then I should see "Thank you"
```

The agent knows to:

- Always add `wait for AJAX to finish` after form submission
- Test both desktop and mobile viewports
- Use the exact field labels from the page
- Handle rate limiting with custom steps when needed
- Follow the `page--test-category.feature` naming convention

How It Works: Every generated line maps to a real built-in step, which is why these files run on the first try. `I fill in "Your Name"` resolves the field by its visible label before falling back to id, name, and placeholder, and the agent used the exact labels it saw while exploring the page, so the label-first match hits. `I select "11-50 employees" from "Team Size"` drives a real select element. The explicit AJAX waits are the readable half of a two-layer defense: underneath, webship-js 2.0's AfterStep hook runs smartSettle after every state-changing step, watching `window.__webshipAjaxCount`, pending timers, and DOM mutations until the page is quiet. The generated suite never needs a sleep tuned to anyone's network speed.

At scale: The naming convention is doing quiet portfolio work. When every agent-built project follows `page--test-category.feature`, the output of `ls tests/features` reads as a coverage map of the site, and the same `ls` means the same thing on all twenty projects. Dashboards can group results by page prefix, and an engineer dropped into an unfamiliar repo knows where everything lives before opening a file.

Recipe 249 Agent Debugging Failed Tests

Problem: Writing tests was never the expensive part, the Wednesday-morning triage is. Three failures in the overnight run: is it the application, the test, or the environment? The engineers who answer that quickly took years to calibrate, and they are spread across too many projects already. The agent ships with that calibration built in.

Solution: When tests fail, the agent does not just report the error. It diagnoses it:

```
Agent output:
"3 of 8 scenarios failed. Analyzing failures:

1. 'Contact form submission' - Step 'I should see Thank you' failed
   Root cause: The site has rate limiting (flood control). After the first
   submission, subsequent attempts within 300 seconds show 'too many
attempts'.
   Fix: Created custom step that accepts either 'Thank you' or rate limit
message.

2. 'Mobile page load' - Step 'I should see Contact' timed out
   Root cause: Mobile viewport triggers lazy loading. Content loads after
   initial networkidle.
   Fix: Added extra 'wait for AJAX to finish' after navigation.

3. 'Links check' - Step 'the info@webship.co link should contain mailto'
   Root cause: Used 'response should contain' which checks text, not HTML
attrs.
   Fix: Changed to 'the "info@webship.co" link should contain "mailto:" by
"href" attr'

Rerunning... 8 of 8 scenarios passed."
```

This is the AI agent at its best, it does not just run tests, it understands why they fail and how to fix them. It knows the difference between a timing issue and a selector issue. It knows that `response should contain` checks text content, not HTML attributes. It knows about rate limiting. All of this domain knowledge is baked into the agent's prompt.

Look at the three failures again and you will see the classic triage taxonomy: an application behavior (flood control), an environment and timing quirk (lazy loading on mobile), and a plain test bug (asserting text where an attribute was needed). Those are the same three buckets your human reviews should sort every failure into, because the response to each is different. An agent that labels the bucket, not just the stack trace, is doing the part of QA work that reports usually leave to whoever has the most scar tissue.

Recipe 250 The Agent Knowledge Base

Problem: A generic AI assistant knows Playwright from its training data, frozen at whenever that snapshot was taken. Ask it for help and it will suggest `waitForTimeout` and hand-rolled CSS selectors for things your framework already ships as one-line steps. The difference between a generic assistant and a specialist is not the model. It is what you put in the prompt.

Solution: The agent's markdown file contains the complete webship-js reference. Here is what it knows:

Agent Knowledge Base

Configuration:
- `cucumber.js structure and all worldParameters`
- `playwright.config.ts settings and browser args`
- Environment variables (BROWSER, HEADLESS, SLOW_MO)
- Project directory structure

Step Definitions (complete reference):
- Navigation (6 steps)
- Form interaction (10 steps)
- Click actions (3 steps)
- Wait and scroll (9 steps)
- Text assertions (6 steps)
- Page assertions (4 steps)
- Element assertions (4 steps)
- Link assertions (2 steps)
- `Checkbox/radio assertions (4 steps)`
- Modal steps (6 steps)
- API testing (22 steps)

Critical Rules:
- `Always wait for AJAX after press/submit`
- Use attribute assertions for HTML attributes
- Handle rate limiting with custom steps
- `Reduce minWaitTime.page for auto-dismissing messages`
- `Use @desktop/@mobile tags for viewport testing`
- `Feature file naming: page--test-category.feature`
- `Custom steps go in tests/step-definitions/custom.js`

World Object API:
- `this.page (Playwright Page)`
- this.context (BrowserContext)
- this.playwrightBrowser (Browser instance)
- this.launchUrl (base URL)
- this.minWaitTime (wait config)
- this.assetsFolder (test assets path)

FIGURE 31.4 *The Agent Knowledge Base (Recipe 250)*

How It Works: This is a curated working set, not a dump. The full framework ships 411 built-in steps across 36 step files; the agent's reference distills the categories that handle most pages, plus the critical rules, the gotchas this book teaches, written as instructions instead of war stories. That curation is deliberate prompt engineering: an agent that knows the 80 steps it needs and the ten mistakes to avoid outperforms one drowning in an unranked list of everything.

At scale: The knowledge base is a file, which means it is extensible. Teams running many projects append their own conventions, the org's tag taxonomy, where selector JSON files live, which environments exist and their base URLs, so the agent writes tests that pass *your* review, not just generic review. Recipe 261 shows the pattern. Once the org's rules live in the agent file, every project gets them automatically; the alternative is re-explaining house style in every session, forever.

Part 2: The Claude Plugin (MCP Server)

What It Is

The `webship-js-claude-plugin` is a Model Context Protocol (MCP) server. MCP is the standard protocol that lets AI tools like Claude Code communicate with external systems. When you install this plugin, Claude gets eight new tools it can call, just like it calls Bash or Read.

The difference between the agent and the plugin:

- **Agent**: Claude becomes a testing specialist (autonomous, full workflow)
- **Plugin**: Claude gets testing tools added to its toolkit (on-demand, per-task)

Recipe 251 Installing the Claude Plugin

Problem: The agent is a specialist you summon for a full job. Some days you want the opposite: you are refactoring a controller and just want to say "rerun the login tests" mid-conversation, without switching context or starting a dedicated testing session. For that, the testing capability has to live quietly inside every conversation, tools, not a persona.

Solution:

```
# Clone and install dependencies
git clone https://github.com/webship/webship-js-claude-plugin.git
cd webship-js-claude-plugin
npm install

# Option 1: Add to Claude Code settings file
# Edit ~/.claude/settings.json or <project>/.claude/settings.json
```

```
{
  "mcpServers": {
    "webship-js": {
      "command": "node",
      "args": ["/path/to/webship-js-claude-plugin/src/index.js"]
    }
  }
}
```

```
# Option 2: Use Claude Code CLI
claude mcp add webship-js node /path/to/webship-js-claude-
plugin/src/index.js
```

How It Works: The settings entry tells Claude Code how to start the server: it spawns `node src/index.js` as a child process and speaks the MCP protocol with it over stdio. From that point the plugin's eight tools sit in Claude's toolkit next to Bash and Read, Claude sees their names, descriptions, and input schemas, and decides when a request calls for them. Register at the project

level (`<project>/.claude/settings.json`) rather than per-user when you work in a team: the configuration ships with the repo, so every engineer who clones it gets the same tools without a setup wiki page.

Recipe 252 The Eight MCP Tools

Problem: Handing an AI write access to your repo goes better when you know exactly what it can touch. Each of these tools maps to a concrete action in your project, a directory created, a file written, a test run, and knowing the inventory is how you predict what a sentence like “set up tests and run the smoke suite” will actually do.

Solution: Once installed, Claude has access to these tools:

MCP Tools Provided by webship-js-claude-plugin

Tool	Description
webshipjs_setup	Create a new test project
Inputs: project_dir, launch_url, min_wait_page	
`Creates: directories, cucumber.js, custom.js boilerplate`	
webshipjs_create_feature	Write a .feature file
Inputs: project_dir, filename, content	
`Creates: tests/features/<filename>`	
webshipjs_create_custom_step	Add a custom step definition
Inputs: project_dir, step_code	
`Appends to: tests/step-definitions/custom.js`	
webshipjs_run_tests	Run the test suite
Inputs: project_dir, tags (optional), timeout	
`Returns: test output (pass/fail)`	
webshipjs_generate_report	Generate HTML report
Inputs: project_dir	
`Returns: summary with pass/fail counts and failure details`	
webshipjs_list_features	List all feature files
Inputs: project_dir	
Returns: file names with scenario counts	
webshipjs_list_steps	Step definition reference
`Inputs: category (navigation/form/click/wait/etc)`	
Returns: all steps in that category	
webshipjs_update_config	`Update cucumber.js settings`
Inputs: project_dir, launch_url, min_wait_page, timeout	
`Updates: cucumber.js configuration`	

FIGURE 31.5 *The Eight MCP Tools (Recipe 252)*

How It Works: Notice what is *not* here: no "deploy", no database access, no tool that reaches outside `project_dir`. Every tool reads or writes ordinary webship-js files, features into `tests/features/`, custom steps appended to `tests/step-definitions/custom.js`, configuration into `cucumber.js`, or shells out to run the suite. That scoping is what makes the plugin safe to leave installed: the blast radius of any tool call is your test directory, and everything it writes shows up in `git diff`. The `tags` input on `webshipjs_run_tests` is the lever you will use most; it is the same tag expression you would pass on the command line.

Recipe 253 Using MCP Tools in Conversation

Problem: There is a person on every team who can describe exactly the test they want, "check that an empty checkout shows the warning, on mobile too", and has never opened `cucumber.js` in their life. Between that sentence and a running test sits the gap where test requests go to die in the backlog. The plugin closes the gap: the description becomes the interface.

Solution: With the plugin installed, you talk to Claude naturally:

```
You: "Set up a test project for https://mystore.com"

Claude calls: webshipjs_setup({
  project_dir: "./test-mystore",
  launch_url: "https://mystore.com",
  min_wait_page: 3000
})

Claude: "Project set up at ./test-mystore. Ready for test creation."
```

```
You: "Create a login test"

Claude calls: webshipjs_create_feature({
  project_dir: "./test-mystore",
  filename: "login--form-validation.feature",
  content: `Feature: Login form validation
  Scenario: Successful login
    Given I am on "/login"
    When I fill in "Email" with "user@example.com"
    And I fill in "Password" with "SecurePass123"
    And I press "Sign In"
    And I wait for AJAX to finish
    Then I should see "Dashboard"`
})
```

```
You: "Run the tests"

Claude calls: webshipjs_run_tests({
  project_dir: "./test-mystore",
  tags: "@desktop"
})
```

```
You: "Show me the report"

Claude calls: webshipjs_generate_report({
  project_dir: "./test-mystore"
})

Claude: "## Test Results
| Metric | Count |
|--------|-------|
| Scenarios | 4 |
| Passed | 3 |
| Failed | 1 |

### Failures
- **Login success**: step 'I should see Dashboard' timed out.
  The login form submits via AJAX but the redirect takes longer..."
```

How It Works: Claude matches your sentence against the tool schemas, “set up a test project” fits `webshipjs_setup`, “run the tests” fits `webshipjs_run_tests`, fills in the arguments, and calls the tool. The detail that keeps this honest: every result lands as an ordinary file in the repo. The feature file Claude “created” is sitting in `tests/features/`, ready for a normal PR review. You get AI convenience without a parallel universe of artifacts that only the AI understands.

At scale: This is the workflow that finally lets product owners and BAs contribute scenarios directly. They describe the behavior in their own words, the plugin writes the feature file, QA reviews the diff like any other change. Sign-off stops being “trust me, it’s green” and becomes a green run of scenarios the PO drafted and can read. Across a portfolio, that quietly fixes the oldest complaint in BDD: acceptance criteria that QA had to rewrite from scratch.

Recipe 254 MCP Resources

Problem: Tools act; resources inform. Claude writing Gherkin mid-conversation is only as accurate as what it remembers about webship-js, and “remembers” is doing dangerous work in that sentence. The plugin solves it by publishing reference material Claude can re-read at any moment instead of guessing.

Solution: The plugin also provides two resources that Claude can read:

```
// Resource: webshipjs://steps/all
// Returns the complete step definition reference

// Resource: webshipjs://tips
// Returns critical tips:
// 1. ALWAYS add "wait for AJAX" after "press Submit"
// 2. Use attribute assertions for HTML attributes
// 3. Handle rate limiting with custom steps
// 4. Reduce minWaitTime.page for auto-dismissing messages
// 5. Use @desktop and @mobile tags
// 6. Feature file naming: page--test-category.feature
// 7. Custom steps in tests/step-definitions/custom.js
// 8. "I fill in" tries: label            ▶ id/name/placeholder
// 9. "I should see" checks visible text; "response should contain" checks textContent
// 10. Each scenario runs in fresh incognito context
```

FIGURE 31.6 *MCP Resources (Recipe 254)*

How It Works: Read the tips list closely and you will recognize this book in miniature, tip 10 is why coupled scenarios fail (and why Recipe 242's advice holds), tip 8 is the label-first field resolution that makes plain-language forms steps work, tip 9 is the text-versus-markup distinction that the agent used to debug the mailto link in Recipe 249. Publishing these as a resource means Claude consults the house rules at the moment of writing, in every conversation, for every engineer, which is more than most teams can say about their own wiki.

Recipe 255 The MCP Server Source Code

Problem: MCP sounds like infrastructure, protocols, servers, capabilities, and teams file it under "someday." Then someone reads the actual source and discovers the entire server is one Node.js file an engineer can absorb in twenty minutes. Before you build your own (Recipe 262), see how little there is to fear.

Solution: The plugin is a single Node.js file that implements the MCP protocol:

```
// src/index.js - simplified structure
import { Server } from '@modelcontextprotocol/sdk/server/index.js';
import { StdioServerTransport } from
'@modelcontextprotocol/sdk/server/stdio.js';

const server = new Server(
  { name: 'webship-js-plugin', version: '2.0.0' },
  { capabilities: { tools: {}, resources: {} } }
);

// Register 8 tools
server.setRequestHandler(ListToolsRequestSchema, async () => ({
  tools: [
    {
      name: 'webshipjs_setup',
      description: 'Set up a new webship-js test project...',
      inputSchema: {
        type: 'object',
        properties: {
          project_dir: { type: 'string' },
          launch_url: { type: 'string' },
          min_wait_page: { type: 'number', default: 3000 },
        },
        required: ['project_dir', 'launch_url'],
      },
    },
    // ... 7 more tools
  ],
}));

// Handle tool calls
server.setRequestHandler(CallToolRequestSchema, async (request) => {
  const { name, arguments: args } = request.params;
  switch (name) {
    case 'webshipjs_setup': return handleSetup(args);
    case 'webshipjs_run_tests': return handleRunTests(args);
    // ...
  }
});
```

```
// Start server
const transport = new StdioServerTransport();
server.connect(transport);
```

This is how you build AI-native testing tools. The MCP protocol is the standard, any AI assistant that supports MCP can use these tools.

How It Works: The stdio transport is the detail that keeps this simple: there is no port to open, no auth to configure, nothing to deploy. Claude Code spawns the process when a session starts and talks to it over stdin/stdout; when the session ends, the process dies. The two request handlers are the whole architecture, one lists what the server offers, one dispatches calls, and the switch statement is the entire routing layer. Anything you can do in Node, you can now hand to an AI as a typed, schema-described tool.

Part 3: Skills (Slash Commands)

What They Are

Skills are the quickest way to invoke webship-js testing in Claude Code. You type a slash command, it expands into a full prompt with context-loading instructions, and Claude executes it.

Recipe 256 Installing Skills

Problem: Some testing actions you trigger twenty times a week: set up a project, test a page, pull a report, look up a step. Typing a paragraph of instructions into Claude each time is friction, and friction is how good practices quietly stop happening. Muscle memory wants a command, not an essay.

Solution:

```
# Clone the repository
git clone https://github.com/webship/webship-js-skills.git

# Option 1: Install globally
cp -r webship-js-skills/.claude/skills/* ~/.claude/skills/

# Option 2: Install to a specific project
cp -r webship-js-skills/.claude/skills/* /path/to/project/.claude/skills/

# Option 3: Use the install script
cd webship-js-skills
bash install.sh              # global
bash install.sh --project /path/to/project
```

How It Works: Each skill is a directory containing a `SKILL.md` file, a named, reusable prompt with instructions and an arguments placeholder. Claude Code picks them up from `~/.claude/skills/` or the project's `.claude/skills/` and exposes each as a slash command. No process, no server: installing a skill is copying a folder.

At scale: Put the skills in your project template repo. Then `/webship-js-test` behaves identically on every project an engineer opens, and onboarding a new QA hire shrinks to "learn four commands" instead of forty wiki pages. The commands become the team's shared vocabulary, when someone says "just run the report skill," everyone on every project knows exactly what that means.

Recipe 257 The Four Skills

Problem: Four commands cover the whole testing lifecycle, but only if you know what each one expects and what it leaves behind. Here is the full set on one page.

Solution:

Available Skills

```
/webship-js-setup <url>
```

Initialize a new webship-js test project.
Creates project structure, installs dependencies,
`configures cucumber.js and playwright.config.ts.`

```
Example: /webship-js-setup https://example.com
```

```
/webship-js-test <path>
```

Create and run BDD tests for a specific page.
Explores the page, writes feature files, runs suite.

```
Example: /webship-js-test /contact
```

```
/webship-js-report [tags]
```

Generate and analyze HTML test reports.
Runs tests, creates report, summarizes results.

```
Example: /webship-js-report @desktop
```

```
/webship-js-steps [category]
```

Quick reference for all step definitions.
Categories: navigation, form, click, wait, assert, etc.

```
Example: /webship-js-steps api
```

FIGURE 31.7 *The Four Skills (Recipe 257)*

Recipe 258 Skill Workflow, From Zero to Tested

Problem: Monday: new client site, zero tests, and a stakeholder who wants “some automated coverage” demonstrated by Friday. The traditional path, scaffold, explore, write, debug, report, is a week you do not have. Here is the same path as five slash commands.

Solution: Here is the complete workflow using only slash commands:

```
# Step 1: Set up a project
/webship-js-setup https://myapp.com

# Claude creates the project, installs dependencies, configures
everything.
# Output: "webship-js project set up at test-myapp-com/"

# Step 2: Test the homepage
/webship-js-test /

# Claude explores the homepage, identifies elements, writes feature files,
# runs the tests. Output: "Created 3 feature files, 6 scenarios. 5 passed,
1 failed."

# Step 3: Test the contact page
/webship-js-test /contact

# Claude does the same for the contact page.

# Step 4: Get a full report
/webship-js-report

# Claude runs all tests and generates an HTML report.

# Step 5: Look up step syntax
/webship-js-steps form

# Claude shows all form-related step definitions.
```

How It Works: Each command expands into a full prompt with context-loading instructions (Recipe 259), so Claude re-grounds itself in the real step definitions before acting. Everything the skills produce is a plain webship-js artifact, feature files, `cucumber.js`, reports, which means

there is no lock-in: stop using the skills tomorrow and the suite keeps running in CI like any hand-written one. The “5 passed, 1 failed” in step 2 is the honest part of the demo; the skill reports what the run reported, and the failure is yours to triage or hand back to Claude.

At scale: This five-command loop is your onboarding path for every new property. The marginal cost of suite number twenty-one is five commands plus a human review of the generated scenarios, which moves the bottleneck where it belongs: deciding what is worth testing, not typing boilerplate. Teams that onboard sites monthly script the sequence into their project kickoff checklist.

Recipe 259 Context Loading in Skills

Problem: The classic AI failure in testing is the step that *almost* exists, a phrasing remembered from training data, one word off from the real one, discovered only when the suite dies with undefined-step errors. Documentation in the model’s memory is documentation that can be stale or scrambled. The cure is procedural: make the AI read the source before it writes a line.

Solution: Each skill has a critical section: **Context Loading**. Before doing any work, the skill instructs Claude to read:

1. **Official documentation**, All webship.co docs pages
2. **Source code**, The actual step definition files (`webship.js`, `webship-api.js`)
3. **Example files**, All 49 example `.feature` files and 30 example HTML pages
4. **Project files**, Existing configuration and tests

This context loading ensures that every skill invocation uses the most accurate, up-to-date step definitions. The source code is the source of truth, not cached documentation, not memory from a previous conversation.

Context Loading Priority

1. Source code (webship.js, webship-api.js) ◀ Source of truth
2. Example .feature files (49 examples) ◀ Patterns
3. Example .html pages (30 pages) ◀ Test targets
4. Official docs (webship.co) ◀ Reference
5. Project files (cucumber.js, existing tests) ◀ Current state

The skill reads ALL of these before writing a single line.
This is why the tests it generates are accurate.

FIGURE 31.8 *Context Loading in Skills (Recipe 259)*

How It Works: The priority order is the point. Step definition files are regular expressions and code, they cannot drift from reality, because they *are* reality. The 49 example feature files show those steps used correctly in context, which teaches the model patterns no API listing can. Docs come fourth because docs describe; source defines. It is the same rule this book keeps repeating about testing itself, applied to the AI: trust what executes, verify everything else.

At scale: Steal this pattern for any in-house AI tooling you build, on any framework. Point the AI at source code and living examples first, documentation second, and its own memory never. One paragraph of context-loading instructions in a prompt file is the difference between an assistant that is confidently wrong about your stack and one that quotes your actual code back to you, and it costs nothing to maintain, because the source maintains itself.

Part 4: Choosing the Right Tool

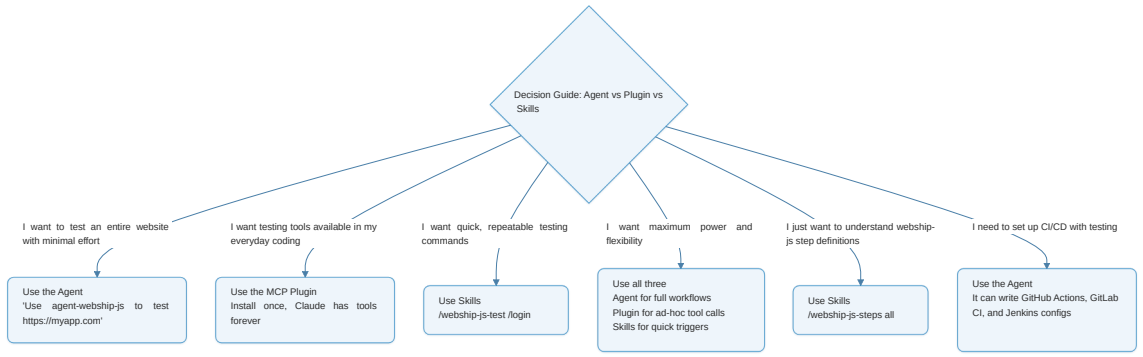


FIGURE 31.9 *Part 4: Choosing the Right Tool*

Part 5: Building Your Own Testing Agent

The webship-js AI agent is open source. You can study its architecture and build agents for your own testing frameworks. Here is the anatomy:

Recipe 260 Agent File Structure

```
---
name: agent-webship-js
description: Automated website testing agent using webship-js
model: opus
tools:
  - Bash
  - Read
  - Write
  - Edit
```

```
  - Glob
  - Grep
  - WebFetch
  - Agent
---

# webship-js Testing Agent

You are an expert automated testing agent specialized in webship-js 2.0...

## Your Capabilities
1. Project Setup
2. Test Creation
3. Custom Steps
4. Test Execution
5. Report Generation
6. Debugging

## Available Step Definitions
### Navigation
- Given I am on "/path"
...

## Critical Rules
1. ALWAYS add wait for AJAX after press
...

## Workflow
1. Explore
2. Plan
3. Create
4. Configure
5. Run
6. Debug
7. Report
```

The key insight: the agent file is a prompt. Everything Claude needs to know about webship-js is in this markdown file. When Claude reads it, it becomes a testing specialist. The better the prompt, the better the agent.

Recipe 261 Extending the Agent

You can customize the agent for your specific application:

```
---
name: agent-myapp-tests
description: Testing agent for MyApp (extends webship-js agent)
model: opus
tools:
  - Bash
  - Read
  - Write
  - Edit
  - Glob
  - Grep
  - WebFetch
---

# MyApp Testing Agent

You are a testing specialist for MyApp, an e-commerce platform.

## Application Knowledge
- Base URL: https://staging.myapp.com
- Admin credentials: admin@myapp.com / TestPass123
- Test database resets every hour

## Custom Test Patterns
- All product pages follow: /products/{slug}
- Cart uses WebSocket updates (wait 2s after adding)
- Checkout requires Stripe test card: 4242424242424242

## Available Steps
(Include all webship-js steps here, plus your custom ones)

## Application-Specific Steps
Given I am logged in as "admin"
Given the cart contains {int} items
When I apply coupon "{string}"
Then the order total should be "{string}"
```

Part 6: Building Your Own MCP Plugin

Recipe 262 MCP Plugin Architecture

The webship-js Claude plugin follows the MCP specification. Here is how to build your own:

```
#!/usr/bin/env node
import { Server } from '@modelcontextprotocol/sdk/server/index.js';
import { StdioServerTransport } from
'@modelcontextprotocol/sdk/server/stdio.js';

const server = new Server(
  { name: 'my-testing-plugin', version: '1.0.0' },
  { capabilities: { tools: {}, resources: {} } }
);

// Define your tools
server.setRequestHandler(ListToolsRequestSchema, async () => ({
  tools: [
    {
      name: 'my_run_tests',
      description: 'Run the test suite with custom configuration',
      inputSchema: {
        type: 'object',
        properties: {
          tags: { type: 'string', description: 'Cucumber tags' },
          browser: { type: 'string', enum: ['chromium', 'firefox',
'webkit'] },
        },
      },
    },
  ],
}));

// Handle tool calls
server.setRequestHandler(CallToolRequestSchema, async (request) => {
```

```
  const { name, arguments: args } = request.params;
  if (name === 'my_run_tests') {
    const cmd = `BROWSER=${args.browser || 'chromium'} npx cucumber-js --
tags "${args.tags || ''}"`;
    const output = execSync(cmd, { encoding: 'utf-8' });
    return { content: [{ type: 'text', text: output }] };
  }
});

// Define resources
server.setRequestHandler(ListResourcesRequestSchema, async () => ({
  resources: [
    {
      uri: 'testing://steps/reference',
      name: 'Step Definition Reference',
      mimeType: 'text/plain',
    },
  ],
}));

const transport = new StdioServerTransport();
server.connect(transport);
```

Part 7: Building Your Own Skills

Recipe 263 Skill File Structure

Skills live in `.claude/skills/<skill-name>/SKILL.md`:

```
---
name: my-testing-skill
description: Run tests for a specific page with viewport checking
---

# /my-testing-skill -- Quick Test a Page

## Arguments
- `$ARGUMENTS` -- The page path to test

## Instructions

1. Read cucumber.js to get the launchUrl
2. Navigate to the page using Playwright MCP browser
3. Identify all testable elements
4. Create feature files with @desktop and @mobile scenarios
5. Run the tests
6. Report results
```

The `$ARGUMENTS` placeholder is replaced with whatever the user types after the slash command. So `/my-testing-skill /login` sets `$ARGUMENTS` to `/login`.

Chapter Summary

- The webship-js ecosystem includes three AI products: Agent, MCP Plugin, and Skills
- The **AI Agent** handles full autonomous testing workflows
- The **MCP Plugin** gives Claude native testing tools via Model Context Protocol
- **Skills** provide quick slash commands for common tasks
- All three share the same knowledge base of webship-js step definitions
- Context loading ensures the AI always uses accurate, up-to-date references
- You can extend or build your own agents, plugins, and skills
- The source code (`webship.js`, `webship-api.js`) is always the source of truth
- AI does not replace the tester, it amplifies the tester's capabilities

Next: Real-World Example, testing webship.co from start to finish.

Chapter 32: Real-World Testing, International and Enterprise Websites

The recipes in this book work for any website. But there is a class of websites that demands extra discipline: international organization sites, government portals, multilingual platforms, and enterprise CMS applications. These sites have unique characteristics, multiple languages, complex navigation hierarchies, accessibility mandates, content approval workflows, and millions of users who depend on them.

I spent years at Vardot testing exactly this class of site, Varbase-powered portals for international organizations, governments, and universities, most of them bilingual or trilingual, many of them RTL. The patterns in this chapter are not theory. They are the ones that survived dozens of real projects, the ones a single QA team can carry from site to site without rewriting everything from scratch.

This chapter takes everything you have learned and applies it to the kind of sites that cannot afford to break. The kind of sites where a broken link, a missing translation, or a failed form costs real credibility, and where the team that finds the break first should be yours, not a visitor's.

The Anatomy of an International Website

Before you write a single scenario, map the territory. Every international site I have tested, whatever the CMS, country, or budget, decomposes into roughly the same anatomy. Treat this map as your test surface: each box is a candidate feature folder, each line inside it a candidate scenario.

Typical International Organization Website

Public-Facing:

- Homepage (multilingual: EN, FR, AR, ES, RU, ZH)
- About Us / Mission
- Programs / Projects
- Publications / Reports (PDF downloads)
- News / Press Releases
- Events / Calendar
- Contact / Office Directory
- Search (global site search)
- Careers / Job Listings
- Data / Statistics / Interactive Dashboards

Content Management:

- Editorial workflow (Draft ▶ Review ▶ Publish)
- Role-based access (Author, Editor, Publisher, Admin)
- Media library (images, documents, videos)
- Taxonomy / Tags / Categories
- Translation management

Technical Requirements:

- WCAG 2.1 AA accessibility compliance
- RTL language support (Arabic, Hebrew)
- Responsive design (mobile-first)
- Performance (CDN, caching, lazy loading)
- Security (CSP headers, cookie consent, data protection)

FIGURE 32.1 *The Anatomy of an International Website*

Notice that the technical requirements at the bottom cut across everything above them. RTL support is not a page, it is a property of every page, and the same goes for accessibility and responsive behaviour. That is why this chapter mixes page-level recipes (homepage, contact, publications) with cross-cutting audits (a11y, mobile, multilingual). Product owners write acceptance criteria for the boxes; your suite also has to test the lines between them.

Testing Strategy: The Three-Tier Approach

Not every test is equally important. When you are testing a large site with hundreds of pages, you need to prioritize ruthlessly. Here is the framework:

```
Three-Tier Testing Strategy

Tier 1: Critical Path (@critical / @smoke)
──────────────────────────────────────────
Run on: EVERY push, EVERY deploy
Tests:
├── Homepage loads in all supported languages
├── Navigation menu works
├── Search returns results
├── Contact form submits
├── Login / authentication works
└── Critical content is visible
Time: < 5 minutes

Tier 2: Functional Integrity (@regression)
──────────────────────────────────────────
Run on: Before staging, before production deploy
Tests:
├── All page types load correctly
├── Forms validate and submit
├── Media embeds work (images, videos)
├── Content management CRUD operations
├── Cross-browser verification
└── API endpoints respond correctly
Time: < 20 minutes

Tier 3: Supplemental (@full / @scheduled)
──────────────────────────────────────────
Run on: Nightly, weekly, or manually
Tests:
├── Accessibility audit (all pages)
├── Multilingual content consistency
├── Performance benchmarks
├── Broken link scanning
├── Edge cases and error handling
└── Visual regression across all breakpoints
Time: < 60 minutes
```

FIGURE 32.2 *Testing Strategy: The Three-Tier Approach*

Each tier answers a different question for a different audience. Tier 1 answers “can we deploy right now?”, it is the gate the pipeline blocks on, and it must stay fast enough that nobody is tempted to skip it. Tier 2 answers “did we break anything we already shipped?”, it runs before a release candidate moves forward, and it is the run a delivery manager wants attached to the release ticket. Tier 3 answers “are we drifting?”, accessibility erosion, translation gaps, performance creep: the regressions no single deploy causes but every deploy contributes to.

At scale: the tags are the contract. When every project in your portfolio uses the same three tags with the same meaning, one question, “is @critical green?”, gets a comparable answer on twenty sites. The moment one project quietly redefines @smoke to include a 12-minute regression pack, your cross-project dashboard starts lying. Write the tier definitions down once, centrally, and treat a tag rename like an API break.

Recipe 264 Testing the Homepage

Problem: It is 8 a.m., the CMS took its security update overnight, and the first email of the day is from a director’s office asking why the homepage shows a maintenance banner. The homepage is the most visited page on any site you run, the first thing stakeholders check, and the last thing anyone consciously tests, everyone assumes someone else looked at it. You need a homepage check that runs on every deploy, in under a minute, without a human in the loop.

Solution:

```
@critical @smoke
Feature: Homepage
  As a visitor
  I want the homepage to load correctly
  So that I can access the website

  Scenario: Homepage loads with core content
    Given I am on the homepage
    And wait for AJAX to finish
    Then I should see "Welcome"
    And I should see "About"
    And I should see "Contact"

  Scenario: Homepage header and footer are present
    Given I am on the homepage
    And wait for AJAX to finish
    Then I should see a "header" element
    And I should see a "footer" element
    And I should see "Privacy Policy"
    And I should see "Terms of Use"

  Scenario: Homepage navigation works
    Given I am on the homepage
    When I click "About"
    And wait for AJAX to finish
    Then I should see "About Us"
    When I move backward one page
    And wait for AJAX to finish
    Then I should see "Welcome"

  Scenario: Homepage search works
    Given I am on the homepage
    When I fill in "Search" with "annual report"
    And I press "Search"
    And wait for AJAX to finish
    Then I should see "Search Results"
    And I should see "Annual Report"
```

How It Works: `Given I am on the homepage` navigates to the `launchUrl` from your world parameters, set it per environment with the `LAUNCH_URL` env var instead of hardcoding it, so the identical feature file tests local, staging, and production. After navigation, webship-js does not trust `load` or `networkidle` alone: its smartSettle routine polls every 50ms until in-flight AJAX (`window.__webshipAjaxCount`) is zero, pending timers are zero, and the DOM has been quiet for at least 250ms, all within a 3000ms per-navigation budget. The explicit `wait for AJAX to finish` step reads that same counter, which makes it a cheap belt-and-braces line on CMS homepages that lazy-load content blocks after first paint. And the `header` / `footer` assertions are structural on purpose: text can survive a broken layout, but a missing `<footer>` element means the page truly fell apart.

At scale: this feature file, structure unchanged, is the first thing to copy into every new project you onboard. Keep the scenario names and the shape identical across the portfolio so a failure on project #14 reads exactly like a failure on project #3, and let the pipeline inject `LAUNCH_URL` per environment. Twenty sites, one homepage pattern, zero per-project config edits: that is the difference between maintaining twenty suites and maintaining one pattern twenty times.

Recipe 265 Testing Multilingual Support

Problem: Your site speaks six languages, and the language switcher is the most-clicked element in the header. Then a deploy ships a routing change, and the French link starts serving English content under French navigation, a half-translated page that nobody on the English-speaking dev team notices for two weeks, until a partner organization does, publicly. For most of your audience the language switcher IS the front door, and it deserves the same automated coverage as login.

Solution:

```
@critical @multilingual
Feature: Language Switching
  As an international visitor
  I want to view the site in my language
  So that I can understand the content
```

```
Scenario: Switch to French
  Given I am on the homepage
  When I click "Français"
  And wait for AJAX to finish
  Then I should see "Bienvenue"
  And I should not see "Welcome"

Scenario: Switch to Arabic (RTL)
  Given I am on the homepage
  When I click "العربية"
  And wait for AJAX to finish
  Then I should see "مرحبا"

Scenario: Switch to Spanish
  Given I am on the homepage
  When I click "Español"
  And wait for AJAX to finish
  Then I should see "Bienvenido"

Scenario: Language persists across pages
  Given I am on the homepage
  When I click "Français"
  And wait for AJAX to finish
  When I click "À propos"
  And wait for AJAX to finish
  Then I should see "À propos de nous"
  And I should not see "About Us"

Scenario: Direct URL for language variant
  Given I am on "/fr"
  And wait for AJAX to finish
  Then I should see "Bienvenue"
  When I go to "/fr/a-propos"
  And wait for AJAX to finish
  Then I should see "À propos"
```

How It Works: the dual assertion in the French scenario is the heart of this recipe: `I should see "Bienvenue"` proves the target language rendered, and `I should not see "Welcome"` proves the source language actually left. Half-translated pages pass the first check and fail the sec-

ond, this pair is your chimera catcher. The Arabic scenario works because Gherkin strings are plain UTF-8: you write العربية directly in the feature file and Playwright matches it in the DOM like any other text, so keep feature files saved as UTF-8 and RTL needs no special handling at the assertion level. The persistence scenario matters most in practice: language switchers typically set a cookie or URL prefix, and the bug class to fear is the language silently resetting on the second click. The direct-URL scenario (`/fr`, `/fr/a-propos`) covers the visitor who lands from a search engine, which is most visitors.

Pro Tip: the moment your multilingual tests go beyond clicking and reading, filling forms on translated pages, remember that `I fill in "Label" with "value"` resolves fields label-first, then by id, name, and placeholder. “Your Name” will not find “Votre nom”. Either keep one feature file per language with translated labels, or target fields `by its "data-test" attribute` so one step phrase works in all six languages.

At scale: languages multiply everything: twenty projects times four languages is eighty homepage variants. Do not write eighty features. Keep one language-switching feature per project, one scenario per language, all tagged `@multilingual`, and let a nightly `--tags "@multilingual"` run sweep the whole matrix. When a translation regression shows up, the scenario name alone tells the delivery manager which site and which language broke, no screenshots required.

Recipe 266 Testing Contact Forms on International Sites

Problem: The contact form is how a ministry, a journalist, or a funding partner reaches the organization, and it keeps growing fields nobody re-tests: organization, country, department, subject classification. The failure mode is quiet. The form still renders, visitors still submit, and a misconfigured required field or a broken spam filter swallows every message for a month. Nobody complains, because the people whose messages vanished have no other channel to complain through.

Solution:

```
@regression @forms
Feature: Contact Form
  As a visitor
```

```
  I want to submit a contact inquiry
  So that I can reach the organization

  Scenario: Contact page shows all required fields
    Given I am on "/contact"
    And wait for AJAX to finish
    Then I should see "Your Name"
    And I should see "Organization"
    And I should see "Email"
    And I should see "Phone"
    And I should see "Subject"
    And I should see "Message"

  Scenario: Empty form shows validation errors
    Given I am on "/contact"
    When I press "Submit"
    And wait for AJAX to finish
    Then I should see "required"

  Scenario: Submit contact form with valid data
    Given I am on "/contact"
    When I fill in "Your Name" with "Dr. Sarah Johnson"
    And I fill in "Organization" with "Research Institute"
    And I fill in "Email" with "s.johnson@research.org"
    And I fill in "Phone" with "+1 (555) 234-5678"
    And I select "Partnership Inquiry" from "Subject"
    And I fill in "Message" with "We would like to explore collaboration
opportunities for the upcoming regional program."
    And I press "Submit"
    And wait for AJAX to finish
    Then I should see "Thank you"

  Scenario: Contact page is accessible
    Given I am on "/contact"
    And wait for the page to load
    Then the page should have no critical accessibility violations
```

How It Works: the first scenario asserts every label before anything is filled, a cheap canary for the "someone edited the form in the CMS" class of regression. The fill steps resolve fields label-first (then id, name, placeholder), exactly the way a human reads a form; if the label-to-input association breaks in the markup, the step fails, and since that same broken association is an accessibility defect, one failure flags two problems. After `I press "Submit"`, webship-js's AfterStep hook runs smartSettle for up to 1500ms, holding the "Thank you" assertion until the submission round-trip and any confirmation rendering go quiet. The closing scenario runs the axe-core audit on the form page deliberately: forms are where critical accessibility violations concentrate, because every input needs a programmatic label.

Handling Rate Limiting

Many enterprise sites implement flood control: after one submission, further attempts within a cooldown window are rejected. In CI this bites on the second run, pipeline A submits the form, pipeline B arrives four minutes later, and the rejection has nothing to do with anyone's code. You cannot assert a single fixed outcome, so write a custom step that accepts either legitimate one:

```
// tests/step-definitions/custom.js
const { Then } = require('@cucumber/cucumber');

Then('I should see a success or rate limit message', async function () {
  const body = await this.page.locator('body');
  const text = await body.textContent();
  const hasSuccess = text.includes('Thank you') || text.includes('Message
sent');
  const hasRateLimit = text.includes('too many') || text.includes('try
again later');

  if (!hasSuccess && !hasRateLimit) {
    throw new Error('Expected success or rate limit message, got
neither');
  }
});
```

```
Scenario: Submit form handles rate limiting gracefully
  Given I am on "/contact"
  When I fill in "Your Name" with "Test User"
  And I fill in "Email" with "test@example.org"
  And I fill in "Message" with "Automated testing inquiry"
  And I press "Submit"
  And wait for AJAX to finish
  Then I should see a success or rate limit message
```

The custom step reads the page body once and passes on either acceptable outcome, success or a flood-control notice, and fails loudly on anything else, so a genuine breakage (a 500 page, a validation dead-end) still goes red.

At scale: flood control is the classic shared-staging trap. When four projects' pipelines hit the same staging server, run B trips the limiter that run A armed, and reviewers start waving away real form failures as "probably the flood control again", the most expensive habit a QA team can develop. The dual-outcome step stops the false reds, but the better fix is environmental: ask the platform team to relax flood control on test environments, and keep the dual-outcome step for production smoke checks where you cannot touch server config.

Recipe 267 Testing Navigation Hierarchies

Problem: A mega menu three levels deep, breadcrumbs, sidebar navigation, a footer with forty links, and the information architect just renamed "Programs" to "What We Do" two sprints before launch. Navigation breaks structurally, not visually: the menu still renders beautifully while half its links point at the old URL structure. A visitor who cannot navigate does not file a bug report. They leave.

Solution:

```
@regression @navigation
Feature: Site Navigation
  As a visitor
```

```
I want to navigate through the site structure
So that I can find the information I need

Scenario: Main navigation links work
  Given I am on the homepage
  When I click "About"
  And wait for AJAX to finish
  Then I should see "About Us"
  When I move backward one page
  And wait for AJAX to finish
  When I click "Programs"
  And wait for AJAX to finish
  Then I should see "Our Programs"
  When I move backward one page
  And wait for AJAX to finish
  When I click "Publications"
  And wait for AJAX to finish
  Then I should see "Publications"

Scenario: Footer links work
  Given I am on the homepage
  When I scroll to bottom
  Then I should see "Privacy Policy"
  And I should see "Terms of Use"
  And I should see "Sitemap"
  When I click "Privacy Policy"
  And wait for AJAX to finish
  Then I should see "Privacy"
  And I should see "Data"

Scenario: Breadcrumb navigation works
  Given I am on "/programs/regional/east-africa"
  And wait for AJAX to finish
  Then I should see "Home"
  And I should see "Programs"
  And I should see "Regional"
  And I should see "East Africa"
  When I click "Programs"
  And wait for AJAX to finish
  Then I should see "Our Programs"
```

How It Works: the main-navigation scenario walks click, assert, back, click again, `I move backward one page` uses browser history instead of re-typing the homepage URL, which is faster and quietly verifies the site survives back-button traffic, something CMS cache layers occasionally mishandle. The breadcrumb scenario starts from a deep URL on purpose: breadcrumbs are generated from the route, and deep links are where route-to-breadcrumb logic breaks first. Between steps, smartSettle keeps assertions from firing while menus are still animating or panels still loading. One detail that saves real debugging time on these sites: when webship-js scrolls an element into view it applies a configurable offset, 60px by default, tunable via `WEBSHIP_SELECTORS_OFFSET`, precisely so sticky headers do not sit on top of the element it is about to interact with.

Pro Tip: when navigation labels differ per language ("Programs", "Programmes", "البرامج"), define named selectors for the menu containers in `tests/selectors/*.json` and assert the structure with CSS, keeping text assertions inside the language-specific features. Structure is language-independent; labels are not.

Recipe 268 Testing Publications and Document Downloads

Problem: For a research organization, publications ARE the product. The annual report launch is coordinated with a press embargo, comms has social posts scheduled for 9:00, and at 8:55 someone discovers the download link returns a 403 because file permissions changed in last night's migration. Listing, metadata, filtering, and the download path all fail independently, so each one needs its own scenario.

Solution:

```
@regression @publications
Feature: Publications
  As a researcher
  I want to access and download publications
  So that I can use them in my work

  Scenario: Publications page lists reports
    Given I am on "/publications"
    And wait for AJAX to finish
    Then I should see "Annual Report"
    And I should see "Policy Brief"
    And I should see "Technical Guide"

  Scenario: Publication detail page shows metadata
    Given I am on "/publications"
    And wait for AJAX to finish
    When I click "Annual Report 2025"
    And wait for AJAX to finish
    Then I should see "Annual Report 2025"
    And I should see "Published:"
    And I should see "PDF"
    And I should see "Download"

  Scenario: Filter publications by type
    Given I am on "/publications"
    And wait for AJAX to finish
    When I select "Policy Brief" from "Type"
    And wait for AJAX to finish
    Then I should see "Policy Brief"
    And I should not see "Technical Guide"

  Scenario: Search publications
    Given I am on "/publications"
    And wait for AJAX to finish
    When I fill in "Search publications" with "climate"
    And wait for AJAX to finish
    Then I should see "Climate Action Report"
```

How It Works: four scenarios, four independent failure modes. The listing scenario proves the index builds. The detail scenario proves the click-through works and the metadata renders, “Published:”, “PDF”, “Download” are the three things a researcher scans for before committing to a click. The filter scenario is the subtle one: `I should not see "Technical Guide"` guards against the classic facet bug where the filter updates the URL but not the result list. CMS publication libraries drive these filters with AJAX, which is why each interaction is followed by the AJAX wait, under the hood webship-js is watching `window.__webshipAjaxCount` drain back to zero before letting the assertion run.

Recipe 269 Testing News and Press Releases

Problem: The news section publishes daily, hourly during a crisis, which is exactly when the communications director is staring at it. Pagination is the silent casualty here: it works on launch day with twelve articles and breaks eight months later when article two hundred pushes a query past a limit nobody knew existed. No human re-tests pagination by hand. Ever.

Solution:

```
@regression @news
Feature: News Section
  As a journalist or visitor
  I want to read the latest news
  So that I stay informed about activities

  Scenario: News page shows recent articles
    Given I am on "/news"
    And wait for AJAX to finish
    Then I should see "Latest News"
    And I should see a ".news-item" element

  Scenario: News article page loads correctly
    Given I am on "/news"
    And wait for AJAX to finish
    When I click "Read more"
    And wait for AJAX to finish
    Then I should see "Published"
    And I should see "Share"

  Scenario: News pagination works
    Given I am on "/news"
    And wait for AJAX to finish
    When I scroll to bottom
    And I click "Next"
    And wait for AJAX to finish
    Then I should see "Latest News"
    And I should see a ".news-item" element
```

How It Works: the listing scenario asserts both text ("Latest News") and structure (a `.news-item` element), the text survives an empty listing, the CSS assertion does not, and an empty listing is precisely the failure you are hunting. `I click "Read more"` deliberately clicks the first matching teaser rather than a hardcoded headline, because headlines change daily and your feature file must not. The pagination scenario scrolls to the bottom first (paginators on news sections often sit below lazy-loaded content), clicks "Next", and re-runs the same structural assertions, page two should look like page one, only older.

Recipe 270 Testing Cookie Consent and Privacy

Problem: Cookie consent is the one feature where a bug is not a bug, it is a regulatory finding. If the banner stops appearing, you are setting analytics cookies without consent for every visitor; if “Save” silently ignores the unchecked Marketing box, you are non-compliant while displaying a UI that promises the opposite. Legal finds out from an external audit, and their first question is “when did this break?”, a question only a test history can answer.

Solution:

```
@critical @privacy
Feature: Cookie Consent
  As a visitor
  I want to manage my cookie preferences
  So that my privacy is respected

  Scenario: Cookie consent banner appears on first visit
    Given I am on the homepage
    Then I should see "cookies"
    And I should see "Accept"

  Scenario: Accept cookies dismisses banner
    Given I am on the homepage
    When I press "Accept"
    And wait for AJAX to finish
    Then I should not see "cookie"

  Scenario: Cookie preferences can be customized
    Given I am on the homepage
    When I press "Manage Preferences"
    And wait for the modal to appear
    Then I should see "Analytics"
    And I should see "Marketing"
    And I should see "Functional"
    When I uncheck "Marketing"
    And I press "Save"
    And wait for AJAX to finish
    Then I should not see "cookie"

  Scenario: Privacy policy is accessible
    Given I am on "/privacy-policy"
    And wait for AJAX to finish
    Then I should see "Privacy Policy"
    And I should see "Data Collected"
    And I should see "Use of the Data"
    And I should see "Cookies"
    And I should see "Changes to the Privacy Policy"
```

How It Works: the first-visit scenario depends on starting with a clean cookie jar, which is what you get, since each scenario runs in a fresh browser context with no stored cookies (to force the state explicitly mid-suite, `Given I am an anonymous user` clears cookies and returns to the launch URL). The preferences scenario is the compliance core: open the modal, uncheck Marketing, save, banner gone. To verify consent beyond the UI, webship-js ships twelve cookie matchers built on Playwright's `context.cookies()`, after saving you can assert `a cookie with the name "consent" should exist`, or that a marketing cookie is absent after opting out. The banner disappearing proves the UI responded; the cookie assertions prove the choice actually stuck.

At scale: privacy regressions are portfolio-wide by nature, because twenty sites usually share one consent module, when it breaks, it breaks everywhere at once. Keep these scenarios in the @critical tier on every project, not the nightly: consent is the one feature where "we caught it within 24 hours" is still an uncomfortable sentence in front of a regulator. A delivery office that can show a green @privacy run per site per deploy has the audit answer ready before the question arrives.

Recipe 271 Testing the Services and Products Section

Problem: Service pages are the pages sales actually sends to prospects, so a broken one fails in front of exactly the audience you most wanted to impress, attached to a proposal email nobody can unsend. They also change rarely, and "rarely" is how content rot sneaks in: an editor archives a page, a link goes orphan, and the 404 sits there until a prospect finds it before you do.

Solution:

```
@regression @services
Feature: Services Page
  As a potential client
  I want to see available services
  So that I can evaluate offerings

  Scenario: Services page lists all offerings
    Given I am on "/services"
    And wait for AJAX to finish
    Then I should see "Services"
    And I should see "Automated Functional Testing Setup"
    And I should see "Configure Remote CI/CD"
    And I should see "Test Scenario Development"

  Scenario: Service detail page works
    Given I am on "/services"
    And wait for AJAX to finish
    When I click "Contact Us"
    And wait for AJAX to finish
    Then I should see "Contact"
    When I move backward one page
    And wait for AJAX to finish
    Then I should see "Services"
```

How It Works: the listing scenario pins the offerings by name, if someone renames or unpublishes “Configure Remote CI/CD” in the CMS, this fails within the hour instead of next quarter. The detail scenario covers the conversion path: services pages exist to produce “Contact Us” clicks, so that click is the behaviour worth testing, and `I move backward one page` confirms a visitor can return to comparison-shopping without losing the listing. The AfterStep settle gives CMS-rendered blocks time to finish mounting before each assertion, so a slow component does not produce a false red.

Recipe 272 Testing Blog and Content Pages

Problem: The blog drives organic traffic, which makes it the section where regressions cost you quietly, not a crash, just an `<article>` tag that stopped wrapping the content after a theme update, taking the structured data and the search ranking with it. Marketing notices the traffic dip in next month's report and asks engineering what changed three weeks ago. Nobody remembers.

Solution:

```
@regression @blog
Feature: Blog Section
  As a reader
  I want to browse and read blog posts
  So that I can learn from the content

  Scenario: Blog listing page loads
    Given I am on "/blog"
    And wait for AJAX to finish
    Then I should see "Blog"

  Scenario: Blog post renders correctly
    Given I am on "/blog"
    And wait for AJAX to finish
    When I click "Read more"
    And wait for AJAX to finish
    Then I should see a "article" element

  Scenario: Documentation page loads
    Given I am on "/docs"
    And wait for AJAX to finish
    Then I should see "Documentation"
```

How It Works: this is deliberately the lightest feature in the chapter, three scenarios, structural assertions, no content specifics. `I should see a "article" element` is the load-bearing line: it asserts the semantic markup survived the theme update, and semantic markup is what search engines and screen readers actually parse. Clicking "Read more" instead of a dated post title keeps

the test stable while posts churn underneath it, the repo rule applies here: test behaviour, not content. Resist the urge to assert headlines on a blog; that is how a team ends up editing feature files every Monday morning.

Recipe 273 Testing Events and Calendar

Problem: Event registration comes with a deadline attached. When the regional conference opens registration and the form is broken, every hour of downtime is a participant emailing the organizers instead, or quietly giving up. Worse, event sections are seasonal: the registration path can sit unexercised for months between conferences, which is exactly long enough for a CMS upgrade to break it with no one watching.

Solution:

```
@regression @events
Feature: Events
  As a participant
  I want to view and register for events
  So that I can attend

  Scenario: Events page shows upcoming events
    Given I am on "/events"
    And wait for AJAX to finish
    Then I should see "Upcoming Events"

  Scenario: Event detail page shows information
    Given I am on "/events"
    And wait for AJAX to finish
    When I click "View Details"
    And wait for AJAX to finish
    Then I should see "Date"
    And I should see "Location"
    And I should see "Register"

  Scenario: Event registration form works
    Given I am on "/events"
    And wait for AJAX to finish
    When I click "View Details"
    And wait for AJAX to finish
    When I press "Register"
    And wait for AJAX to finish
    Then I should see "Registration"
    When I fill in "Full Name" with "Dr. Maria Santos"
    And I fill in "Email" with "m.santos@university.edu"
    And I fill in "Organization" with "University of Geneva"
    And I press "Submit Registration"
    And wait for AJAX to finish
    Then I should see "Registration Confirmed"
```

How It Works: the three scenarios form a funnel: the listing renders, the detail page shows the three facts a participant needs (date, location, how to register), and the registration form takes a submission end to end. The registration scenario chains two click-throughs before any filling, each followed by the AJAX wait so multi-step page loads settle fully, smartSettle holds each step until pending requests and DOM mutations go quiet. The fill steps resolve “Full Name”, “Email”, and “Organization” by visible label first, so the test fails when the label-input association fails: a free accessibility check on the form most likely to be rebuilt for every event. Run this funnel on a schedule even between events, against a staging event kept permanently open for exactly this purpose.

Recipe 274 Testing Search Across the Entire Site

Problem: On a thousand-page site, search is not a feature, it is the primary navigation. It is also the feature with the most moving parts: an index that must rebuild as content changes, a query pipeline, facets, language analyzers. When the indexer silently stops, search still “works”, it just returns last month’s site. No error, no crash, and nobody inside the organization notices, because staff navigate by muscle memory, not by search.

Solution:

```
@critical @search
Feature: Global Search
  As a visitor
  I want to search the entire site
  So that I can find specific content quickly

  Scenario: Search returns relevant results
    Given I am on the homepage
    When I fill in "Search" with "annual report 2025"
    And I press "Search"
    And wait for AJAX to finish
    Then I should see "Search Results"
    And I should see "Annual Report"

  Scenario: Empty search shows appropriate message
    Given I am on the homepage
    When I press "Search"
    And wait for AJAX to finish
    Then I should see "enter a search term"

  Scenario: Search with no results shows message
    Given I am on the homepage
    When I fill in "Search" with "xyznonexistentterm123"
    And I press "Search"
    And wait for AJAX to finish
    Then I should see "No results found"

  Scenario: Search filters work
    Given I am on the homepage
    When I fill in "Search" with "policy"
    And I press "Search"
    And wait for AJAX to finish
    And I select "Publications" from "Content Type"
    And wait for AJAX to finish
    Then I should see "Policy"
```

How It Works: four scenarios cover the four states every search has, results, empty query, no results, filtered results. The happy path asserts a specific known document ("Annual Report") rather than settling for "Search Results", and that specificity is what catches the stale-index failure: a generic results page renders fine on a dead index. The empty and no-results scenarios pin the guidance messages, because a blank results page reads as a crash to a visitor. The filter scenario chains a facet selection after the initial query and waits out the AJAX round-trip, facets are almost always asynchronous, and webship-js releases the step only when its request counter (`window.__webshipAjaxCount`) drains to zero.

At scale: search infrastructure is usually shared, one Solr or Elasticsearch cluster behind many sites, so a search failure is rarely one project's failure. When the nightly `@search` runs go red on six projects at once, the pattern is the diagnosis: look at the cluster, not the sites. This is one of the strongest arguments for stamping reports with metadata (`--env`, `--app-version`, `--metadata Key=Value` on the report generator), so the platform team can line failures up across projects by time and environment instead of opening twenty HTML files.

Recipe 275 Testing Careers and Job Listings

Problem: The careers portal has a peculiar audience: people you want to hire, actively evaluating you. A dead "View Position" link or an empty location filter tells a strong candidate more about your engineering culture than the job ad does. And listings churn constantly, HR posts and closes positions weekly through the CMS, so this section breaks far more often than its traffic suggests.

Solution:

```
@regression @careers
Feature: Careers
  As a job seeker
  I want to browse and apply for positions
  So that I can find employment opportunities

  Scenario: Careers page lists open positions
    Given I am on "/careers"
    And wait for AJAX to finish
    Then I should see "Career Opportunities"

  Scenario: Job listing shows details
    Given I am on "/careers"
    And wait for AJAX to finish
    When I click "View Position"
    And wait for AJAX to finish
    Then I should see "Job Description"
    And I should see "Requirements"
    And I should see "Apply"

  Scenario: Filter jobs by location
    Given I am on "/careers"
    And wait for AJAX to finish
    When I select "Geneva" from "Location"
    And wait for AJAX to finish
    Then I should see "Geneva"
```

How It Works: the scenarios assert the stable skeleton and ignore the churn: the page frames its purpose ("Career Opportunities"), a click-through reveals the three blocks every posting needs (description, requirements, a way to apply), and the location filter narrows. Notice what is NOT asserted, no job titles, no listing counts, because content that changes weekly would turn this into a feature file someone has to edit weekly. The filter scenario's closing `I should see "Geneva"` is intentionally loose for the same reason; tighten it only if your filter develops a history of lying.

Recipe 276 Testing Data and Statistics Dashboards

Problem: The statistics dashboard is the page your organization cites in press releases, and the hardest page you own to test. It renders client-side from API calls, so the page can "load" perfectly while every chart sits empty because one endpoint timed out. Layout checks pass, visitors see no data, and the data team hears about it when a journalist asks why the dashboard is blank.

Solution:

```
@regression @data
Feature: Data Dashboard
  As a researcher
  I want to access interactive data
  So that I can analyze trends

  Scenario: Dashboard page loads with visualizations
    Given I am on "/data"
    And wait for AJAX to finish
    Then I should see "Dashboard"
    And I should see a ".chart-container" element

  Scenario: Dashboard filters update data
    Given I am on "/data"
    And wait for AJAX to finish
    When I select "2025" from "Year"
    And wait for AJAX to finish
    Then I should see "2025"

  Scenario: Data export works
    Given I am on "/data"
    And wait for AJAX to finish
    Then I should see "Download"
    And I should see "CSV"
```

How It Works: asserting `I should see a ".chart-container" element` is the right altitude for canvas- and SVG-rendered charts: you cannot cheaply assert what a chart draws, but you can assert that its container mounted, which fails the moment the rendering JavaScript throws. This is

also where webship-js's settle logic earns its keep: dashboards fire several API calls and animate on arrival, and smartSettle holds each step until in-flight requests and DOM mutations go quiet instead of asserting against a half-drawn page. Pair this feature with JS error capture, `WEBSHIP_JS_ERROR_MODE=fail` suite-wide, or the `@js-fail` tag on just this feature, because on a dashboard a console exception usually IS the bug, even when the page looks superficially fine.

Pro Tip: for deterministic dashboard tests, take the data warehouse out of the equation: `Given the URL "**/api/stats" returns the JSON:` intercepts matching requests through Playwright's `page.route()` glob matching and serves your fixture instead. The filter scenario then tests the dashboard's behaviour, not the warehouse's mood. Keep one unmocked scenario as a live-data canary.

Recipe 277 API Testing for International Sites

Problem: The same content feeds the website, the mobile app, and three partner organizations syndicating your publications, all through one API. Browser tests will never catch the field that quietly disappeared from the JSON payload, because the website's frontend was updated to match. The mobile app was not. Its users get a crash, and the app-store reviews arrive before the bug report does.

Solution:

```
@regression @api
Feature: Public API
  Background:
    Given the API base URL is "https://api.example.org"

  Scenario: API returns list of publications
    When I send a GET request to "/api/publications"
    Then the API response code should be 200
    And the response should be valid JSON
    And the JSON response should have property "data"
```

```
Scenario: API returns single publication
  When I send a GET request to "/api/publications/1"
  Then the API response code should be 200
  And the JSON response should have property "title"
  And the JSON response should have property "published_date"
  And the JSON response should have property "language"

Scenario: API supports language filter
  When I send a GET request to "/api/publications?lang=fr"
  Then the API response code should be 200
  And the response should be valid JSON

Scenario: API returns events
  When I send a GET request to "/api/events"
  Then the API response code should be 200
  And the response should be valid JSON

Scenario: API returns countries data
  When I send a GET request to "/api/countries"
  Then the API response code should be 200
  And the response should be valid JSON
  And the JSON response should have property "data"

Scenario: API 404 for unknown resource
  When I send a GET request to "/api/nonexistent"
  Then the API response code should be 404

Scenario: API with authentication
  Given I am authenticating as "api_user" with "api_key_123" password
  When I send a GET request to "/api/admin/stats"
  Then the API response code should be 200
  And the JSON response should have property "total_visitors"
```

How It Works: `Given the API base URL is "..."` in the Background sets the base for every request in the feature, and the requests go through axios rather than the browser, so these scenarios run in milliseconds with no page involved. The property assertions are contract checks: `the JSON response should have property "title"` fails the moment a serializer change drops a field a consumer depends on, whether or not the website still renders. The language-filter

scenario (`?lang=fr`) is the international-site special, translated content endpoints break independently of the default language, and only an explicit per-language request catches it. The 404 scenario pins error behaviour, because an API that answers unknown routes with a 200 HTML error page (a classic CMS misconfiguration) breaks every consumer that parses JSON. The authenticated scenario proves the wall stands in one direction; pair it with an unauthenticated request to the same admin endpoint asserting a 401 or 403 to prove it stands in the other.

At scale: API features are the cheapest tests in your portfolio, no browser, no screenshots, milliseconds per scenario, so they are the tier you can run on every push across twenty projects without anyone asking about CI minutes. Keep the Background's base URL as the single per-environment line the pipeline owns, and tag the suite `@api` so it can run standalone (`npx cucumber-js --tags "@api"`) as a fast first gate before any browser container spins up.

Recipe 278 Accessibility Audit for International Sites

Problem: For public institutions, WCAG 2.1 AA is procurement criteria, legislation, or both, and the accessibility statement on your site is a claim auditors test. The cruel part is how accessibility regresses: one unlabeled input this sprint, one low-contrast button the next, each individually invisible, until the annual external audit drops a forty-page findings document on the delivery manager's desk and every line item becomes rush-priority at once.

Solution:

```
@a11y @scheduled
Feature: Accessibility Compliance
  Scenario: Homepage is accessible
    Given I am on the homepage
    And wait for the page to load
    Then the page should have no critical accessibility violations

  Scenario: Contact page is accessible
    Given I am on "/contact"
    And wait for the page to load
    Then the page should have no critical accessibility violations

  Scenario: Publications page is accessible
    Given I am on "/publications"
    And wait for AJAX to finish
    And wait for the page to load
    Then the page should have no critical accessibility violations

  Scenario: News page is accessible
    Given I am on "/news"
    And wait for AJAX to finish
    And wait for the page to load
    Then the page should have no critical accessibility violations

  Scenario: Keyboard navigation works on homepage
    Given I am on the homepage
    When I press the "Tab" key
    Then the focused element should be visible
    When I press the "Tab" key
    Then the focused element should be visible
    When I press the "Enter" key
    And wait for AJAX to finish
    Then I should not see "error"
```

How It Works: `the page should have no critical accessibility violations` injects axe-core into the live page and runs its rule engine against the rendered DOM, real markup, real contrast, real ARIA relationships, not a static lint of templates. Critical violations (missing form la-

bels, broken ARIA references, keyboard traps) are the ones that make a page unusable with assistive technology, which is why this threshold makes a sane CI gate where a full audit would drown you in noise; when you are ready for the stricter bar, the same engine backs `the page should pass an accessibility audit at level "AA"`. The keyboard scenario covers what no rule engine can: whether tabbing actually moves focus somewhere visible. And the `wait for the page to load` lines before each audit are not decoration, auditing a page whose components are still mounting produces phantom violations, and settling first prevents exactly that.

Pro Tip: when an audit fails in CI, reproduce locally and add `Then print accessibility violations` after the failing step, you get the violation list with selectors in your console instead of reverse-engineering it from a report.

At scale: run the `@a11y` suite nightly on every project and aggregate the results in one place, this is the quality signal management actually understands, because "zero critical violations across all twenty sites" maps directly onto the compliance language they report upward. Stamp each run with the report generator's metadata flags (`--env`, `--app-version`, `--metadata Project=name`) so nightly reports are comparable across the portfolio, and treat a new critical violation like a failed build rather than a backlog item. The whole point is to never again meet accessibility debt forty pages at a time.

Recipe 279 Testing Mobile Responsiveness

Problem: In many of the regions international organizations serve, mobile is not a viewport, it is the internet. A contact form that works on desktop but hides its Submit button behind a sticky footer at 375px wide is, for a majority of your real audience, simply broken. Yet every developer and every stakeholder reviews the site on a large monitor, so mobile regressions sail through every human check you have.

Solution:

```
@mobile @regression
Feature: Mobile Experience
  Scenario: Homepage loads on mobile
    Given I set the viewport to mobile
    And I am on the homepage
    And wait for AJAX to finish
    Then I should see "Welcome"

  Scenario: Mobile navigation menu works
    Given I set the viewport to mobile
    And I am on the homepage
    And wait for AJAX to finish
    When I click "Menu"
    And wait for AJAX to finish
    Then I should see "About"
    And I should see "Programs"
    And I should see "Contact"
    When I click "Contact"
    And wait for AJAX to finish
    Then I should see "Contact"

  Scenario: Contact form works on mobile
    Given I set the viewport to mobile
    And I am on "/contact"
    And wait for AJAX to finish
    When I fill in "Your Name" with "Mobile User"
    And I fill in "Email" with "mobile@example.com"
    And I fill in "Message" with "Testing from mobile viewport"
    And I press "Submit"
    And wait for AJAX to finish
    Then I should see "Thank you"

  Scenario: Tables are readable on mobile
    Given I set the viewport to mobile
    And I am on "/data"
    And wait for AJAX to finish
    Then I should see "Dashboard"
```

How It Works: `I set the viewport to mobile` resizes the page to a phone-sized viewport, and everything after it runs the same steps as desktop, that symmetry is the point. webship-js carries a named breakpoint registry for exactly this work (xs at 375x667 up through xxxl at 1920x1080, with xl at 1200x900 as the default), and you can redefine the whole set as JSON via `WEBSHIP_SELECTORS_BREAKPOINTS`. The menu scenario exercises the hamburger pattern, navigation collapsed behind a "Menu" toggle, and the form scenario re-runs the full fill-and-submit flow because mobile is where elements actually overlap: Playwright refuses to click an element that another element covers, so a sticky banner sitting on the Submit button fails here while passing on desktop. That refusal is not flakiness. It is the bug you wanted caught.

At scale: breakpoints are a portfolio convention, not a per-project setting. Define them once, the same `WEBSHIP_SELECTORS_BREAKPOINTS` JSON exported by your shared CI configuration, so "mobile" means 375x667 on every project and a designer's "which width failed?" has one answer everywhere. And do not double your whole suite for mobile: run everything at desktop, plus a curated `@mobile` subset (navigation, forms, anything sticky-positioned) per project. That covers the genuinely viewport-sensitive behaviour for roughly 15% of the runtime, a number your DevOps team will accept without a meeting.

Complete Project Structure for an International Site

Here is how the whole chapter lands on disk. Two conventions carry this structure: feature files are named `area--behaviour.feature`, so a failing filename alone tells you what broke before you open anything, and folders mirror the site's information architecture, so a product owner can find the scenarios for "their" section without a guided tour.

```
organization-tests/
├── package.json
├── cucumber.js
├── playwright.config.ts
│
├── tests/
│   ├── features/
│   │   ├── homepage/
│   │   │   ├── homepage--page-load.feature
│   │   │   ├── homepage--navigation.feature
│   │   │   └── homepage--search.feature
│   │   ├── contact/
│   │   │   ├── contact--page-load.feature
│   │   │   ├── contact--form-validation.feature
│   │   │   └── contact--form-submission.feature
│   │   ├── publications/
│   │   │   ├── publications--listing.feature
│   │   │   └── publications--filters.feature
│   │   ├── news/
│   │   │   └── news--listing.feature
│   │   ├── multilingual/
│   │   │   └── language-switching.feature
│   │   ├── privacy/
│   │   │   ├── cookie-consent.feature
│   │   │   └── privacy-policy.feature
│   │   ├── api/
│   │   │   ├── publications-api.feature
│   │   │   └── events-api.feature
│   │   └── accessibility/
│   │       └── a11y-audit.feature
│   │
│   ├── step-definitions/
│   │   └── custom.js
│   │
│   └── support/
│       └── hooks.js
│
├── tests/reports/
└── .github/workflows/test.yml
```

FIGURE 32.3 *Complete Project Structure for an International Site*

Keep `custom.js` honest: it should hold only steps genuinely specific to this site, like the rate-limit step from Recipe 266. The moment a custom step proves useful on a second project, move it out of the project and into a shared package, copy-pasted step definitions are how twenty suites drift into twenty dialects.

Running the Suite with Tags

Tags are the API of your test suite, they are how pipelines, schedules, and humans select what runs. These commands assume the taxonomy from the three-tier strategy:

```
# Critical path -- run on every push
npx cucumber-js --tags "@critical"

# Full regression -- run before deploy
npx cucumber-js --tags "not @scheduled and not @wip"

# Accessibility audit -- run nightly
npx cucumber-js --tags "@a11y"

# Mobile tests only
npx cucumber-js --tags "@mobile"

# Multilingual tests
npx cucumber-js --tags "@multilingual"

# API tests (no browser needed)
npx cucumber-js --tags "@api"

# All tests, parallel execution
npx cucumber-js --parallel 4 --tags "not @wip"
```

At scale: standardize this exact command set across every project. When all pipelines in the portfolio call the same handful of commands with the same tags, the platform team maintains one CI template instead of twenty bespoke pipelines, and webship-js ships starter configs for thirteen CI providers (GitHub Actions, GitLab CI, Jenkins, Azure Pipelines, and friends), so the template is mostly written for you. The `--parallel 4` flag is your nightly cost lever: workers multiply CPU but divide wall-clock time, and on a farm running forty suites a night, wall-clock is what you pay for.

The Testing Advantage for International Organizations

If you ever need to justify the testing budget to a steering committee, this is the slide:

Why Automated Testing Matters for International Sites

Without automated tests:

- A broken translation goes unnoticed for weeks
- A form stops working after a CMS update
- Cookie consent breaks and violates GDPR
- Navigation links point to 404 pages
- Mobile users cannot access critical services
- Accessibility regressions exclude disabled users

With automated tests:

- Every page is verified on every deploy
- Forms are tested with valid and invalid data
- Privacy compliance is continuously verified
- Navigation is tested across all languages
- Mobile experience is verified automatically
- Accessibility is audited regularly

The cost of a bug on a site serving millions of people
is not measured in dollars. It is measured in trust.

Automated testing is not a luxury for international sites.
It is the foundation of reliability.

FIGURE 32.4 *The Testing Advantage for International Organizations*

Chapter Summary

- International websites have unique requirements: multilingual, accessible, secure, high-traffic
- Use a three-tier strategy: Critical (every push), Regression (before deploy), Supplemental (nightly)
- Test the homepage first, it is the front door
- Test language switching and content persistence across navigation
- Handle rate limiting with custom steps that accept multiple outcomes
- Test cookie consent for privacy compliance
- Verify accessibility with axe-core integration
- Test mobile responsiveness, mobile is the primary device in many regions
- API tests complement browser tests for data endpoints
- Use tags to organize and run test subsets efficiently
- The cost of a bug on an international site is measured in trust, not just money

Chapter 33: The Complete Step Definition Reference with Examples

Step definitions and code examples by Tasneem Natshah, from the webship-js source.

webship-js 2.0 ships 411 built-in step definitions across 36 step files, demonstrated by 49 example feature files and 30 example HTML pages in the repo. This chapter walks through every major step pattern, including the attribute-based variants that many teams miss. Consider this the encyclopedia of webship-js steps, the chapter you keep open in a second tab while you write features.

Every phrasing in this chapter was checked against the 2.0 step sources. Where 1.x habits no longer match 2.0 behavior, especially around waits, the text says so explicitly.

Navigation Steps, All Variants

Recipe 280 All Navigation Patterns

Problem: You onboard your third project of the quarter and find three different navigation habits in its features: full URLs hardcoded in some scenarios, relative paths in others, and one file that navigates by clicking through menus because nobody trusted the URL steps. The suite passes against staging and fails against the review environment, because half the URLs point at the wrong host. Navigation is the first thing every scenario does, if it is inconsistent, everything downstream is.

Solution:

```
# Navigate to the base URL
Given I am on the homepage
Given I am on homepage

# Navigate to a specific path (appended to launchUrl)
Given I am on "/about-us"
Given I am on "/products/laptop-pro"

# Navigate during the test
When I go to the homepage
When I go to "/contact"

# Browser history
When I move backward one page
When I move forward one page

# Page reload
When I reload the page
```

How it works under the hood:

```
// When I go to "/about-us"
// 1. If the argument starts with http:// or https://, navigate directly
// 2. Otherwise, prepend launchUrl: http://localhost:8080/about-us
// 3. Wait for <body> to attach, then run smartSettle:
//    domcontentloaded -> networkidle -> ajax 0, timers 0, DOM quiet 250ms
await gotoUrl(this.page, this.launchUrl + url);
await this.page.waitForSelector('body', { state: 'attached', timeout:
defaultTime });
await waitForPageLoad(this.page, defaultTime);  // alias for smartSettle
```

Three details matter here. First, the base URL comes from `worldParameters.launchUrl`, which reads the `LAUNCH_URL` environment variable and falls back to `http://localhost:8080`. Your features never contain a hostname. Second, navigation does not stop at Playwright's `networkidle`, after the load events, smartSettle keeps polling until in-flight fetch/XHR requests reach zero, pending timers reach zero, and the DOM has been quiet for at least 250ms. Third, the whole settle is bounded by `minWaitTime.page` (3000ms by default), so a page with a runaway polling script cannot stall the step forever. There is also `Given I am an anonymous user`, which clears all cookies in the browser context before going to the launch URL, the cleanest way to guarantee a logged-out starting state.

At scale: When one team runs suites against 20 sites, `LAUNCH_URL` is the difference between a reusable suite and 20 forked ones. CI sets `LAUNCH_URL=https://staging.project-a.example` per pipeline; developers set it in `.env` for local DDEV instances; nobody ever edits `cucumber.js` per project. The moment you see a hardcoded hostname in a feature file during review, send it back, that is the seed of an unmaintainable fleet.

Form Steps, All Variants

Recipe 281 Filling Fields by Label, Placeholder, and Name

Problem: A new hire opens their first PR with twelve scenarios, and every field lookup uses a CSS ID copied from DevTools. Two sprints later the front-end team renames those IDs during a component refactor and all twelve scenarios fail at once, on a form whose behavior never changed. The fix is to target fields the way a user would: by what they can read on the screen.

Solution: The `I fill in` step is the most versatile step in webship-js. It tries multiple strategies to find the field:

```
# By visible label text
When I fill in "Username" with "john_doe"

# By placeholder text
When I fill in "Enter your email" with "john@example.com"

# By name attribute
When I fill in "password" with "SecurePass123"
```

How it works:

```
// The fillField helper resolves the field in strict order:
// 1. page.getByLabel(field, { exact: true })        -> the associated
input
// 2. page.getByPlaceholder(field, { exact: true })  -> placeholder match
// 3. page.locator('[name="field"]')                 -> name attribute
fallback
//
// The buildSelector helper (used by the "by attr" variants) is broader:
// - starts with #, ., or [ and has no space -> used as-is (raw CSS)
// - bare token  -> candidate list: tag, #id, .class, [name], [data-
testid],
//   [data-test-id], [data-test], [data-cy], [aria-label], [value],
//   [placeholder], [title]
// - text with spaces -> [value], [placeholder], [aria-label], [title]
// - explicit attribute -> [attr="value" i]  (the "i" = case-insensitive)
```

The resolution is label-first and exact-match. That is deliberate: an exact `getByLabel` match is the same lookup a screen reader performs, so a passing fill step doubles as a weak accessibility check, if webship-js can find your field by label, so can assistive technology. When no strategy matches, the step fails with a friendly error that lists exactly what it tried and suggests the `by attr` variant, so the person reading the CI log knows the next move without opening the source.

At scale: Label-first targeting is what lets one suite pattern survive across many projects. Markup changes constantly; the words on the form change rarely, and when they do, a product person notices. One caution for multilingual portfolios: labels are translated strings, so suites that run against localized environments should either pin the test language or switch those fields to the attribute variants from Recipe 282.

Recipe 282 Filling Fields by Attribute

Problem: The design system team ships a beautiful new search component, and its input has no `<label>`, a placeholder that changes with every marketing review, and a generated class name. Label-first lookup has nothing to hold on to. You need a way to target the field that survives both copy edits and CSS refactors.

Solution: When labels are missing or ambiguous, use attribute-based selectors:

```
# By any attribute (auto-detected: ID, class, or name)
When I fill in "user_email" for "#email-input" by attr

# By specific attribute
When I fill in "John Smith" for "uname" by its "id" attribute

# By placeholder attribute
When I fill in "Dr. Smith" for "Your full name" by its "placeholder"
attribute
```

How It Works: The `by attr` form (no attribute named) runs the value through `buildSelector`, which builds a candidate list covering `#id`, `.class`, `[name]`, and the whole test-attribute family, `data-testid`, `data-test-id`, `data-test`, and `data-cy`, plus `aria-label`, `value`, `placeholder`, and `title`. The first match on the page wins. The explicit form (`by its "id" attribute`) is stricter: it builds `[id="value" i]`, with the `i` flag making the match case-insensitive, so `"UName"` and `"uname"` both hit. Note the argument order: the value to type comes first, the target second, `I fill in "<value>" for "<target>"`.

At scale: Pick one test-attribute convention, `data-test` is a common choice, and write it into the front-end standards of every project your team owns. webship-js already looks for it, so there is nothing to configure on the testing side. The payoff shows up on the third project you onboard: QA stops negotiating selectors with each front-end team separately, and a step like `fill in "x" for "search-input" by its "data-test" attribute` reads the same in every repo.

Recipe 283 Filling Multiple Fields with a Table

Problem: The registration form has eleven fields. Written as eleven `When I fill in` lines, the scenario is so long that reviewers scroll past it without reading, and the one field with a wrong value hides in the noise. You want the form data to read like a form, a compact block a product owner can scan in five seconds.

Solution:

```
# Standard table fill (by label/name)
When I fill in the following:
  | Username | john_doe    |
  | Password | SecurePass  |
  | Email    | j@test.com  |

# Table fill by attribute
When I fill in the following: by attr
  | #uname   | John Smith |
  | pwordcss | 1234       |

# Table fill by specific attribute
When I fill in the following: by its "placeholder" attribute
  | Your full name | John Smith |
  | Your Password  | 1234       |
```

How It Works: Each table row runs through the same field-resolution pipeline as the single-field step, label, placeholder, then name for the standard form; `buildSelector` for the `by attr` forms. The whole table is one Cucumber step, which has a quiet performance benefit in 2.0: the auto-settle hook that runs after every state-changing step fires once for the table instead of once per field. On a form whose fields trigger per-keystroke validation requests, that is ten settles you did not pay for. The table also makes the failure message precise, webship-js tells you which row failed, not just “the step failed.”

Clearing a Field

To clear a field without typing anything, use the `with:` variant with no value:

```
# Clear a field (empty it)
When I fill in "Search" with:
When I fill in "#email" with: by its "id" attribute
```

The `with:` variant fills the field with an empty string, Playwright's `fill('')`, which clears existing content and fires the same input events a user's select-all-and-delete would. Useful before re-typing into a field that a previous step or the application itself pre-populated.

Recipe 284 Pressing Buttons, All Variants

Problem: It is the Tuesday before release and the checkout suite fails on one machine but not another. The culprit is the step after the button press: the click fired, the request went out, and the next assertion ran before the response landed. Button presses are where state changes, which makes them the place where timing discipline matters most.

Solution:

```
# By visible text or value
When I press "Submit"
When I press "Login"
When I press "Add to Cart"

# By attribute (auto-detected)
When I press "#btn-submit" by attr
When I press "btn-login" by attr

# By specific attribute
When I press "btn-pressid" by "id" attr

# Slow endpoint? Give the settle a bigger budget after the press
When I press "Submit"
And I wait for AJAX to finish
```

How It Works: `I press` matches buttons by visible text or `value` attribute; the `by attr` variants route through `buildSelector` exactly like the fill steps. Here is what changed in 2.0: you no longer need to chase every press with a wait. An `AfterStep` hook runs `smartSettle(page, 1500)` after every state-changing step, which waits for in-flight fetch/XHR requests, pending timers, and DOM mutations to go quiet before the next step starts. The explicit `I wait for AJAX to finish` still exists, it runs the same settle with a 10-second

budget, so reach for it when the press triggers something slower than 1.5 seconds, like a payment authorization or a report generation. If you need to opt out of auto-settling entirely (say, to assert on a loading spinner), set `WEBSHIP_AUTO_SETTLE=off`.

At scale: Audit your older suites for the 1.x habit of `press` + `wait for AJAX to finish` on every interaction. Across 40 nightly suites, those redundant 10-second-budget waits are mostly free (they return early on settle), but the noise hides the few places where the bigger budget is genuinely needed. Keep the explicit wait only where a comment can say why, "report generation takes 4-6s", and let the AfterStep hook handle the rest.

Recipe 285 Select, Check, Uncheck, Radio

Problem: Dropdowns, checkboxes, and radios are where generic click steps betray you. Clicking a `<select>` opens a native widget Playwright cannot see into; clicking a styled checkbox often hits the decorative `<span>` instead of the input. You need steps that speak the form controls' own language.

Solution:

```
# Select from dropdown
When I select "United States" from "Country"
When I select "Premium Plan" from "Subscription"

# Additionally select (multi-select)
When I additionally select "Canada" from "Countries"

# Checkboxes
When I check "I agree to the terms"
When I uncheck "Send me newsletters"

# By attribute
When I check "#newsletter"

# Radio buttons
When I select radio button "Male"
When I select radio button "Express Shipping"

# File upload
When I attach the file "test-document.pdf" to "#file-upload"
```

How It Works: `I select ... from ...` uses Playwright's `selectOption`, which works on the real `<option>` elements and fires a proper `change` event. The detail teams miss: `I additionally select` adds to the current selection of a multi-select **without clearing it**, plain `I select` replaces the selection. Use the pair deliberately: `select` for the first value, `additionally select` for each one after. `check` / `uncheck` and `select radio button` accept a visible label, a value, an id, or a class, so they keep working when the design system hides the native input behind a styled element. `attach the file` resolves the path relative to your test assets and sets it straight on the `<input type="file">`, no OS file dialog is ever opened, which is why file uploads are not flaky in webship-js.

Click Steps, All Variants

Recipe 286 Clicking Links and Buttons

Problem: The admin screen lists fifty users, each row with its own “Edit” link. `When I click "Edit"` finds fifty matches and clicks the first one, which is not the user your scenario created. Sooner or later every team rediscovers this and reaches for `nth-child` selectors, and the test breaks the next time the table is re-sorted. Scoped clicks fix the ambiguity without coordinates.

Solution:

```
# Click by visible text
When I click "About Us"
When I click "Read More"
When I click "Home"

# Click by attribute (auto-detected)
When I click "#aboutUsid" by attr

# Click by specific attribute
When I click "aboutUsid" by its "id" attribute
When I click "aboutUs" by "class" attr

# Click within a table row
When I click "Edit" in the "John Smith" row
When I click "Delete" in the "Expired Product" row
When I click "View" in the "john@example.com" row

# Click within modals
When I click "Confirm" in the modal
When I click "Save" button in the modal
When I close the modal
```

How It Works: The row-scoped variant (`click "Edit" in the "John Smith" row`) first finds the `<tr>` whose text contains the identifier, then searches for the clickable text inside that row only. Pick the most unique cell value as the identifier, an email or an ID column beats a first name. The modal-scoped variants do the same trick with the dialog: webship-js resolves the modal container (a named `modal` selector if you registered one, otherwise ARIA `role="dialog"` and native `<dialog>` elements) and clicks within it, so a "Confirm" button in the page background cannot steal the click. For role-based precision there is also the web-first family, `When I click the "Save" button`, `When I click the "Pricing" link`, and the other `getByRole` variants, which targets elements by their accessibility role and polls up to 5 seconds for the element to become actionable.

At scale: Row-scoped and modal-scoped clicks are the steps to push hardest in cross-team reviews, because they are the ones that keep table-heavy admin UIs testable without per-project step code. A useful convention for the portfolio: scenario data should always create or reference a uniquely identifiable row (email, reference number), never "the first row", ordering is an implementation detail that varies per environment and per dataset.

Wait Steps, All Variants

Recipe 287 Every Wait Pattern

Problem: Your flakiest suite has fourteen sleeps in it. Each one was added on a bad afternoon to make one failure go away, each one slows every run, and none of them documents what it is actually waiting for. The team is afraid to remove them. Before you can clean that up, you need to know what each wait step in webship-js 2.0 really does, because most of them no longer sleep at all.

Solution:

```
# Settle on demand: in-flight requests, pending timers, DOM mutations
And I wait for AJAX to finish
And wait for AJAX to finish

# Settle with the page-load probes (body attached -> domcontentloaded
# -> networkidle -> quiet DOM)
And I wait until the page is loaded
And we wait until the page is loaded

# Wait UP TO a time budget -- returns early the moment the page settles
And I wait 1 second
And I wait 2 seconds
And I wait 3s
And wait 1s
And wait 5s

# Same bounded settle, alternate phrasing
And I wait max of 3 seconds

# Minute-scale budgets (long-running operations, imports, batch jobs)
And I wait 1 minute
And I wait 2 minutes
And I wait max of 1 minute

# Wait for modal
And wait for the modal to appear
And wait for the modal to disappear
```

2.0 also ships event waits, name the condition instead of guessing a duration:

```
# Wait for an element or text to arrive / leave
When I wait for "#results" to appear
When I wait for ".spinner" to disappear
When I wait for the text "Saved" to appear
When I wait for the text "Processing" to disappear

# Wait for navigation and page state
When I wait until the URL contains "/dashboard"
When I wait until the page title contains "Dashboard"
When I wait until the page is interactive

# Wait for counts, network, timers
When I wait until 5 elements match ".search-result"
When I wait until at least 3 elements match ".card"
When I wait until the network is idle
When I wait until pending timers settle

# Polling assertion with its own deadline
Then eventually I should see "Import complete" within 30 seconds
```

When to use which:

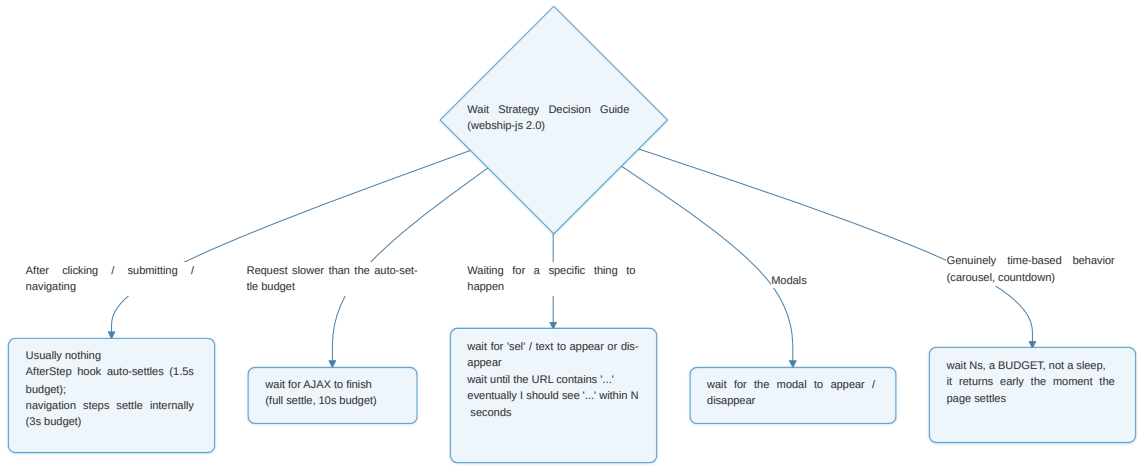


FIGURE 33.1 *Every Wait Pattern (Recipe 287)*

Rule of thumb: name the event you are waiting for. A duration in a feature file should be the exception that carries a comment.

How It Works: Every wait above funnels into one function: `smartSettle(page, timeout)`. At context startup, webship-js injects instrumentation via `addInitScript`: a fetch/XHR wrapper that maintains `window.__webshipAjaxCount`, a `setTimeout` wrapper that maintains `window.__webshipPendingTimers`, and a `MutationObserver` that stamps `window.__webshipLastMutation`. “Settled” means all three agree: zero in-flight requests, zero pending timers, and no DOM mutation for at least 250ms, checked every 50ms after the body-attached, `domcontentloaded`, and `networkidle` probes pass. So `I wait 5 seconds` does not sleep for five seconds; it polls that condition with a 5-second deadline and returns the instant the page goes quiet. The repo calls this the Behavior-Based Robotics principle: react to the environment, not the clock. On top of that, an `AfterStep` hook runs `smartSettle(page, 1500)` after every state-changing step (disable with `WEBSHIP_AUTO_SETTLE=off`), which is why most 2.0 scenarios contain no wait steps at all.

At scale: Do the math for your CI farm. A suite with fourteen 3-second sleeps wastes up to 42 seconds per run; across 40 suites running nightly on three browsers, that is over an hour of compute a day spent waiting for nothing. Because 2.0 timed waits return early, simply upgrading recovers most of that, but the better move is a portfolio-wide review rule: any `wait N seconds` line needs a comment naming the time-based behavior it covers, or it gets replaced with an event wait.

Scroll Steps, All Variants

Recipe 288 Every Scroll Pattern

Problem: The footer links never fail locally, but in CI the page is taller than the viewport and the lazy-loaded testimonials section between the hero and the footer never initializes, because nothing ever scrolled past it. Lazy loading, infinite scroll, and sticky headers all make scroll position part

of your application's behavior, which means your tests sometimes need to control it.

Solution:

```
# Directional scroll
When I scroll down
When I scroll up
When I scroll left
When I scroll right

# Scroll by pixel amount
When I scroll down 500
When I scroll up 200
When I scroll left 300
When I scroll right 300

# Scroll to extremes
When I scroll to the top
When I scroll to the bottom
When I scroll to the start
When I scroll to the end

# Scroll to a specific element
When I scroll to top of "#content-section"
When I scroll to bottom of "#content-container"
```

How It Works: Directional scrolls without a number move one viewport-sized step; with a number they move exactly that many pixels. The element-targeted variants scroll until the element's top or bottom edge enters view, handy when a sticky header would otherwise cover the thing you are about to click. The part that makes scrolling reliable in 2.0: whatever the scroll triggers, an IntersectionObserver firing a fetch, a lazy image swap, an infinite-scroll page append, shows up in `__webshipAjaxCount` and the mutation timestamp, so the auto-settle after the scroll step waits for the loaded content before your next assertion runs. You rarely need `scroll down 500` + `wait 2 seconds` pairs anymore; the settle is built in.

Assertion Steps, All Variants

Recipe 289 Text Assertions

Problem: `Then I should see "Error"` passes, because the word “Error” appears in a help-center link in the footer, not because your form showed the validation message. Whole-page text assertions are the most-used and most-abused steps in any suite. The fix is scoping: assert the text where it is supposed to appear.

Solution:

```
# Text is visible on page
Then I should see "Welcome to our site"
Then I should see "Login successful"

# Text is NOT visible
Then I should not see "Error"
Then I should not see "Access Denied"

# Text matches a regex pattern
Then I should see text matching "Order #[0-9]+"
Then I should see text matching "\\d{4}-\\d{2}-\\d{2}"

# Text inside a specific element
Then I should see "John Smith" in the "user-name" element
Then I should see "Active" in the "status" element

# Text inside element by attribute
Then I should see "user1" in the "uname" element by its "id" attr
Then I should see "1234" in the "pwordcss" element by attr

# Text inside a table row
Then I should see "Active" in the "John Smith" row
Then I should see "$999.99" in the "Laptop Pro" row

# Text matches pattern inside element
Then I should see text matching "Total: \\$[0-9.]+" in the "cart-total"
element
```

How It Works: The plain form checks the rendered text of the whole page, text a user could actually read, not markup. The element-scoped forms narrow the search to one container resolved by `buildSelector`, the row-scoped form narrows it to the matching `<tr>`, and `text matching` compiles your pattern as a JavaScript regex (escape backslashes once for Gherkin, hence `\\d`). Scoping is not just about correctness; it is about failure messages. “Expected ‘Active’ in the ‘status’ element” points a reviewer at one element. “Expected ‘Active’ on the page” points them at the

whole DOM. For assertions that should tolerate slow rendering, the web-first family (`Then "sel" should have text "..."` / `should contain text`) polls every 100ms for up to 5 seconds before failing.

At scale: Page-level `I should see` is fine for smoke tests; element-scoped assertions are the standard to hold in shared suites. The portfolio failure mode is real: sites share headers, footers, cookie banners, and chat widgets full of words like “Error”, “Success”, and “Submit”. A whole-page assertion that passes for the wrong reason on project A gets copied to project B, where it fails for an even more confusing reason. Scope early.

Recipe 290 Page and URL Assertions

Problem: The login test clicks “Sign in” and asserts “Welcome”, which also appears on the marketing page the app redirects to when authentication silently fails. Text can lie about where you are. The URL cannot.

Solution:

```
# Current URL matches
Then I should be on "/dashboard"
Then I should be on the homepage

# URL matches a pattern
Then the url should match "/products/.*"

# Response contains text (checks textContent, not HTML attributes)
Then the response should contain "Welcome"

# HTTP status code
Then the response status code should be 200
```

How It Works: `I should be on "/path"` compares the current path against your expectation, with the base URL stripped, so the same assertion holds in every environment. `the url should match` takes a regex, which is what you want for URLs containing generated IDs (`/orders/[0-9]+/confirmation`). Two traps worth naming: `the response should contain` checks ren-

dered text content, not raw HTML, attribute values like `mailto:` links will not match (see Recipe 292). And `the response status code` refers to the main page response, not API calls made by the page; for those, use the API steps (Recipes 295-297) or network recording.

Recipe 291 Element Assertions

Problem: A deploy goes out and the hero banner simply is not there, no error, no broken text, just a missing component. Text assertions cannot catch what was never rendered, and a human only notices when a customer does. Element assertions are how a suite notices absence.

Solution:

```
# Element exists on page
Then I should see a "header" element
Then I should see a "nav" element
Then I should see a ".sidebar" element

# Element exists by attribute
Then I should see a "submit-btn" element by "id" attr

# Element does NOT exist
Then I should not see a "error-banner" element
Then I should not see a "modal" element by "class" attr

# Field contains value
Then the "Username" field should contain "john_doe"
Then the "Email" field should not contain "invalid"

# CSS property assertion
Then the "header" element should contain "background-color:rgb(26, 82,
118)"
Then the "error" element should contain "color:red"
Then the "header" element should not contain "display:none"
```

CSS Color Normalization: webship-js automatically normalizes colors for comparison. When you write `color:red`, it converts both the expected and actual values to RGB format before comparing. This means you can write colors in any format:

```
# All of these work -- webship-js normalizes to RGB internally
Then the "title" element should contain "color:rgb(26, 82, 118)"
Then the "title" element should contain "color:#1a5276"
Then the "alert" element should contain "color:red"
```

Recipe 292 Link Assertions

```
# Link text has expected href
Then the "About Us" link should contain "/about"
Then the "Home" link should contain "/"

# Link by attribute has expected href
Then the "aboutUsid" link should contain "/about-us" by "href" attr

# IMPORTANT: Use attribute assertions for email/tel links
# BAD (checks text content, not HTML attributes):
# Then the response should contain "mailto:info@example.com"

# GOOD (checks the actual href attribute):
Then the "info@example.com" link should contain "mailto:" by "href" attr
Then the "+1-555-0100" link should contain "tel:" by "href" attr
```

Recipe 293 Checkbox and Radio Assertions

```
# Checkbox is checked
Then the "newsletter" checkbox should be checked
Then the "#newsletter" checkbox should be checked

# Checkbox is NOT checked
Then the "marketing" checkbox should not be checked

# Radio button is selected
Then the radio button "Express" should be selected
Then the "shipping-method" radio button is selected
```

Modal Steps, All Variants

Recipe 294 Complete Modal Reference

```
# Modal visibility
Then I should see a modal
Then I should not see a modal

# Modal with specific title
Then I should see a modal with title "Confirm Delete"

# Text inside modal
Then I should see "Are you sure?" in the modal
Then I should see "This action cannot be undone" in the modal

# Click inside modal
When I click "Confirm" in the modal
When I press "Save Changes" in the modal
When I click "Cancel" in the modal content

# Fill fields inside modal
When I fill in "Name" with "New Value" in the modal

# Close modal
When I close the modal

# Wait for modal
And wait for the modal to appear
And wait for the modal to close
```

API Steps, Complete Reference

Recipe 295 Setup Steps

```
# Base URL
Given the API base URL is "https://api.example.com"

# Authentication
Given I am authenticating as "admin" with "password123" password

# Headers - single
Given I set header "Content-Type" with value "application/json"
Given I set the header "Authorization" to "Bearer token123"
Given the header "Accept" is "application/json"

# Headers - multiple
Given I set the following headers:
  | Content-Type  | application/json |
  | Authorization | Bearer token123  |
  | Accept        | application/json |

# Request body - inline JSON
Given I set the request body to '{"name": "John", "email":
"john@example.com"}'

# Request body - table
Given I set the request body with:
  | name  | John Doe         |
  | email | john@example.com |
  | age   | 30               |

# Placeholders
Given I set placeholder "{{userId}}" to "123"
Given I set placeholder "{{baseUrl}}" to "https://api.example.com"
```

Recipe 296 Request Steps

```
# Basic requests (all HTTP methods)
When I send a GET request to "/users"
When I send a POST request to "/users"
When I send a PUT request to "/users/1"
When I send a PATCH request to "/users/1"
When I send a DELETE request to "/users/1"

# POST with doc string body
When I send a POST request to "/users" with body:
  """
  {
    "name": "John Doe",
    "email": "john@example.com"
  }
  """

# POST with values table
When I send a POST request to "/users" with values:
  | name  | John Doe         |
  | email | john@example.com |
  | age   | 30               |

# POST with form data
When I send a POST request to "/login" with form data:
  """
  username=admin
  password=secret
  remember=true
  """

# Using placeholders in requests
Given I set placeholder "{{postId}}" to "1"
When I send a GET request to "/posts/{{postId}}"
```

Recipe 297 Response Assertion Steps

```
# Status code
Then the API response code should be 200
Then the API response code should be 201
Then the API response code should be 404

# Response body contains text
Then the API response should contain "John Doe"
Then the API response should not contain "password"

# Valid JSON
Then the response should be valid JSON

# JSON property exists
Then the JSON response should have property "id"
Then the JSON response should have property "address.city"
Then the JSON response should not have property "password"

# JSON property value
Then the JSON property "userId" should be 1
Then the JSON property "name" should be "John Doe"
Then the JSON property "id" equal to 1

# JSON structure match
Then the API response should contain json:
  """
  {
    "title": "Test Post",
    "userId": 1
  }
  """

# Response header
Then the response header "Content-Type" should be "application/json"

# Debug output
Then print API response
```

Dynamic Content Patterns

Recipe 298 Testing AJAX-Loaded Content

webship-js ships with detailed examples for dynamic content. These patterns are essential for modern web applications:

```
Feature: AJAX Content Loading
  Scenario: Wait for search results
    Given I am on "/search"
    When I fill in "laptop" for "#search-input" by attr
    And I press "Search"
    And I wait for AJAX to finish
    Then I should see "Search Results"
    And I should see "Laptop Pro 15"

  Scenario: Wait for form submission response
    Given I am on "/contact"
    When I fill in "John Doe" for "#name-input" by attr
    And I fill in "john@example.com" for "#email-input" by attr
    And I press "Submit"
    And I wait for AJAX to finish
    Then I should see "Form submitted successfully"
    And I should see "Thank you, John Doe"
```

Recipe 299 Testing Dynamically Rendered Elements

Elements that appear after JavaScript execution:

```
Feature: Dynamic Elements
  Scenario: Click dynamically loaded button
    Given I am on "/dynamic-page"
    When I wait 2 seconds
    Then I should see "Dynamic Content Loaded"
    When I press "Submit Form"
    Then I should see "Form submitted successfully"

  Scenario: Fill dynamically rendered form
    Given I am on "/dynamic-page"
    When I click "Show Registration Form"
    And I wait 1 second
    When I fill in "john.doe@example.com" for "#email" by attr
    And I fill in "John Doe" for "#fullname" by attr
    And I fill in "SecurePass123" for "#password" by attr
    And I press "Register"
    Then I should see "Registration successful"

  Scenario: Select from dynamically populated dropdown
    Given I am on "/dynamic-page"
    When I scroll down 200
    And I wait 1 second
    When I click "Load Countries"
    And I wait 2 seconds
    When I select "United States" from "country"
    Then I should see "You selected: United States"

  Scenario: Interact with dynamically loaded table
    Given I am on "/dynamic-page"
    When I click "Load Users Table"
    And I wait 2 seconds
    Then I should see "John Smith" in the "john@example.com" row
    When I click "Edit" in the "john@example.com" row
    Then I should see "Editing user: john@example.com"
```

```
Scenario: Check dynamically rendered checkboxes
  Given I am on "/dynamic-page"
  When I click "Show Preferences"
  And I wait 1 second
  When I check "#newsletter"
  And I check "#notifications"
  Then the "#newsletter" checkbox should be checked
  And the "#notifications" checkbox should be checked
```

The Behat Migration: API Steps Origin

The API step definitions in webship-js have an interesting history. They were originally written in PHP for Behat's WebAPIExtension. When that project was archived, every step definition was re-built in Node.js for webship-js.

The mapping is direct:

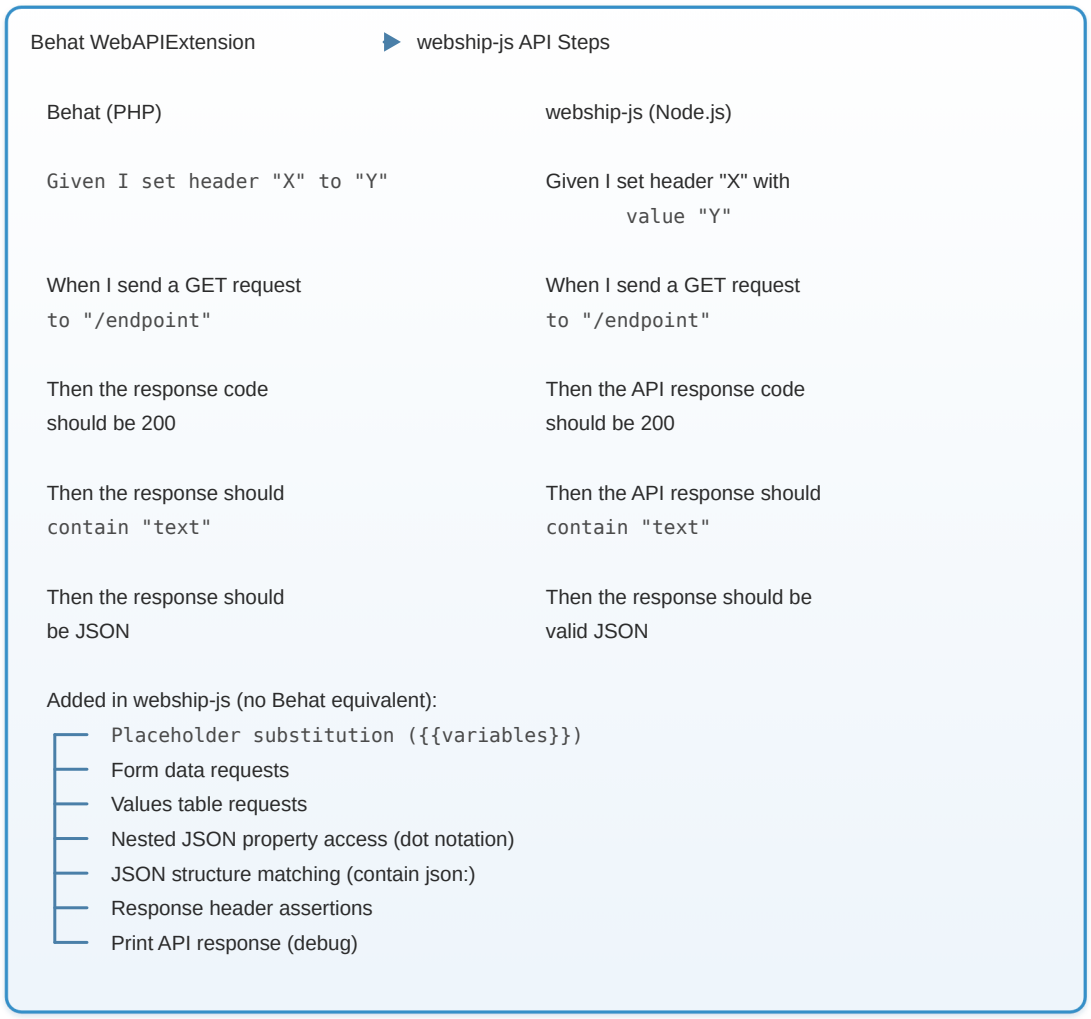


FIGURE 33.2 *The Behat Migration: API Steps Origin*

Teams migrating from Behat to webship-js will find the API steps familiar, the Gherkin is nearly identical.

Screenshot Steps

Recipe 331 Manual Screenshot Capture

```
# Save the current viewport
When I save screenshot
When save screenshot
When we save screenshot

# Save the full scrollable area
Then I save fullscreen screenshot

# Resize then capture
Then I save 1440 x 900 screenshot
Then I save fullscreen 1920 x 1080 screenshot

# Explicit filename (tokens supported)
Then I save screenshot with name "homepage.png"
Then I save screenshot with name "hero-{datetime}.png"
Then I save screenshot with name "{feature_file}_{step_line}"
Then I save fullscreen screenshot with name "about-full.png"
```

Filename tokens:

TOKEN	EXAMPLE VALUE
`{datetime}`	`2025-03-14T10-32-05`
`{feature_file}`	`login`
`{step_line}`	`42`
`{url_path}`	`about-us`
`{failed_prefix}`	`failed_` (on failed steps)
`{ext}`	`png`

Configure via `worldParameters.screenshot.*` or the `WEBSHIP_SCREENSHOT_*` env vars: `DIR`, `PURGE`, `ON_FAILED` (default true), `ON_EVERY_STEP`, `FULLSCREEN`, `PATTERN`, `PATTERN_FAIL`, `FAILED_PREFIX`, `INFO_TYPES`.

On failure, webship-js auto-saves a `failed_<datetime>.<feature>.feature_<line>.png` next to your normal screenshots. No extra step needed.

Named Selector Steps

Recipe 332 Full Selector-System Reference

```
# Register at runtime
When I add "cta" selector for ".hero .btn-primary" css selector
When I add "admin link" selector for "//a[contains(.,'Admin')]" xpath
```

```
selector

# Bulk definition via table
Given I define css selectors:
  | header | header.site-header |
  | nav    | nav[role=navigation] |
  | footer | footer.site-footer |

Given I define xpath selectors:
  | first heading | //h1[1] |
  | login link    | //a[contains(.,'Login')] |

# Load from JSON file
When I add selectors from "homepage-selectors.json" file

# Debug
Then I print css selectors
Then I print xpath selectors

# Visibility
Then I see visible header, nav, footer
Then I don't see modal, overlay

# Focus
Then I see cta has focus
When I move focus to "Email" field

# Text selection
When I select all text in "Title" field
When I select from 0 to 5 text in "Body" field
When I select "admin" text in "Username" field

# Click by name
When I click cta
When I click header, nav
When I click on logo
```

Selector JSON structure:

```
{
  "css":   { "hero": ".hero-banner", "cta": ".hero .btn" },
  "xpath": { "page title": "//h1[contains(@class,'title')]" }
}
```

Recipe 333 Spatial Relation Assertions

```
Then I see header above footer
Then I see nav below header
Then I see logo to the left of nav
Then I see nav to the right of logo
Then I see logo inside of header
Then I see page outside of content
Then I see modal over main
Then I see header not over content

# Multi-subject / multi-target (Cartesian dispatch)
Then I see heading, subline above list, footer
```

Uses Playwright's `boundingBox()` plus computed `z-index` (for `over` / `not over`).

Recipe 334 Named Breakpoints

```
Given I am viewing the site on a xl screen
Given I am viewing the site on a "xxl" screen
Given I am viewing the site on a md device
```

Built-in names:

NAME	WIDTH X HEIGHT	TARGET
xs	375 x 667	phone portrait
sm	576 x 800	large phone / phablet
md	768 x 1024	tablet portrait
lg	992 x 768	laptop
xl	1200 x 900	desktop (default)
xxl	1400 x 900	wide desktop
xxxl	1920 x 1080	Full HD

Override via `worldParameters.selectors.breakpoints` or `WEBSHIP_SELECTORS_BREAKPOINTS` (JSON string).

Chapter Summary

- webship-js ~2.0 ships 96+ built-in step definitions across 14 categories: navigation, forms, waits, clicks, visibility, tables, API, screenshots, selectors, spatial, breakpoints, focus, click-by-name, and more.
- The `buildSelector` helper tries multiple strategies to find elements.
- Attribute-based steps (`by attr`, `by its "id" attribute`) handle cases where labels are ambiguous.
- Table-based filling (`I fill in the following:`) reduces repetition in form tests.
- AJAX wait is the most critical step, use it after every form submission and navigation.
- `response should contain` checks text content; use `by "href" attr` for HTML attributes.
- All API steps were rebuilt from Behat's WebAPIExtension with additional capabilities.
- Screenshot steps support manual capture, resized capture, filename tokens, and auto-capture on failure.
- Named selectors (css + xpath) plus spatial assertions turn tests into layout documentation.
- Named breakpoints (xs -> xxxl) make responsive testing concise.
- The 49 example feature files and 33 HTML pages in the webship-js repo serve as a living reference.

Part VIII: Expanded Ecosystem

Chapter 34: Advanced Selectors, Named Locators, Spatial Relations, Viewports

Let me paint a picture. Your feature files are readable today. `When I click "Login"` tells any stakeholder what happens. Then the designer rebuilds the login button into a custom component that no longer accepts a visible label. Now you need `nav.primary a[data-role='auth-cta']:not(.disabled)`. One CSS selector, buried in one test, unreadable to anyone who doesn't read the DOM.

Multiply by 50 tests. Then multiply by the eight other projects your team maintains, each with its own copy of that selector, each drifting at its own pace. The readability you started with is gone, and so is any hope of fixing a markup change in one place.

Webship-js ~2.0 solves this with a **named selector system**. You register long CSS or XPath strings once with a friendly name, then refer to them in tests by that name. You also get spatial assertions (`above`, `below`, `inside`, `over`), viewport breakpoints by name (`xs`, `md`, `xl`), and focus / text-selection steps that work on any labeled field.

The whole family lives in `webship-selectors.js`. It is loaded automatically in the default `cucumber.js`.

The Named Selector System

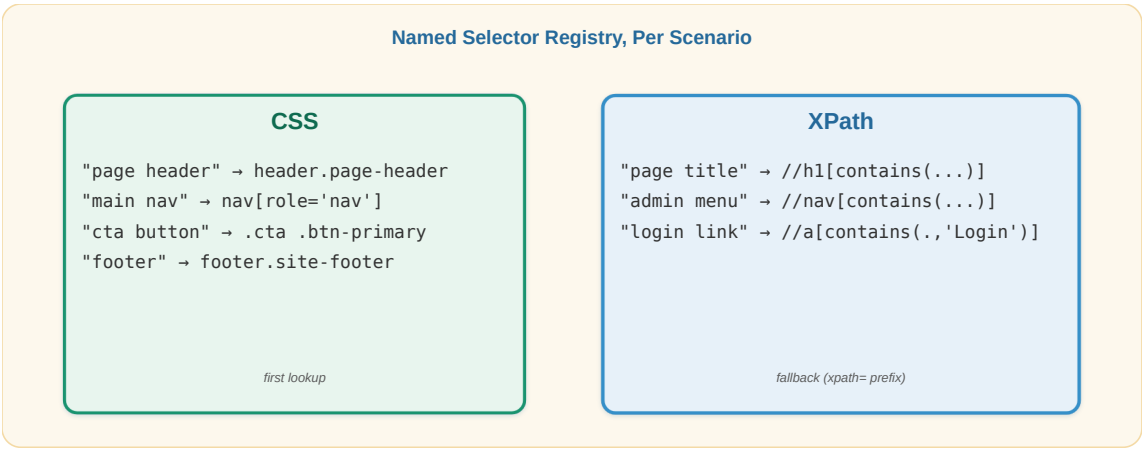


FIGURE 34.1 *The Named Selector System*

Every scenario starts with a fresh copy of the registry. A `Before` hook rebuilds it from `worldParameters.selectors`, the inline `css` and `xpath` maps first, then every JSON file listed in `files` (read from `filesPath`, which defaults to `./tests/selectors/`), in order. Later entries override earlier ones with the same name, **last wins**. This lets each scenario tweak selectors without leaking to the next.

The same `selectors` block carries two more settings you will meet later in this chapter: a scroll offset (60px by default, tunable with `WEBSHIP_SELECTORS_OFFSET`, headroom for sticky headers when elements get scrolled into position) and the named viewport breakpoints (`WEBSHIP_SELECTORS_BREAKPOINTS`).

Recipe 310 Registering a Named Selector Inline

Problem

The front-end team ships a redesign on Thursday. The “Add to cart” button is now an icon-only custom component, no visible label, no stable text for a click-by-name step to find. The quick fix is pasting `.product__buy button[type=submit]` straight into the feature file, and that is the moment your Gherkin stops being readable by anyone outside the QA team. You need the raw CSS. You just don’t want it living in the scenario.

Solution

```
Feature: Cart test with named selectors

  Scenario: Add to cart
    Given I am on "/products/101"
    When I add "add-to-cart" selector for ".product__buy
button[type=submit]" css selector
    And I click add-to-cart
    Then I see visible add-to-cart
```

The step writes `add-to-cart` into the scenario’s CSS registry, the same per-scenario map the `Before` hook seeds from `worldParameters` and your JSON files. Registration lasts for this scenario only; the next scenario gets a fresh copy, so nothing leaks between tests. Every later step that takes a name (`I click ...`, `I see visible ...`, the spatial assertions) resolves it through the registry: CSS map first, XPath map as the fallback. Register the same name twice and the last registration wins, handy when one scenario needs to shadow a shared name without touching the shared file. When the CSS changes, you edit one line.

XPath variant

```
Scenario: Click via XPath
  Given I am on "/dashboard"
  When I add "admin link" selector for "//a[contains(normalize-
space(.),'Admin')]" xpath selector
  And I click admin link
```

The step automatically prefixes `xpath=` when routing to Playwright's locator API, so the expression is evaluated as XPath instead of CSS. Same registry rules apply: scenario-scoped, last wins. One note on resolution order, if the same name exists in both registries, the CSS entry wins, because lookup always tries the CSS map first.

Recipe 311 Defining Selectors in Bulk

Problem

Your layout scenario needs ten selectors, and the inline `add` step works, once per selector. Now the preamble is longer than the test, and the reviewer on your PR scrolls past ten lines of plumbing to find the three assertions that actually matter. The new hire copies that pattern into the next feature, and the noise compounds from there.

Solution

```
Scenario: Landing page layout
  Given I am on homepage
  And I define css selectors:
    | topbar  | .topbar                |
    | header  | header.site            |
    | nav     | nav[role=navigation]   |
    | main    | main                   |
    | footer  | footer                 |
    | cta     | .hero .cta-primary     |
  Then I see visible topbar, header, nav, main, footer
  And I see header above main
  And I see main above footer
```

The `define css selectors:` step takes a Gherkin data table. Each row is a `name | selector` pair that lands in the same per-scenario registry as the inline `add` step, one write per row, last write wins, so a duplicate name later in the table (or in a later step) overrides the earlier one. For distinct names, order does not matter. There is a side benefit nobody plans for: the table reads like a glossary. A reviewer can sanity-check the names against the design system without parsing a single CSS rule.

XPath bulk variant

```
Given I define xpath selectors:
  | first heading | //h1[1]                              |
  | login link    | //a[contains(normalize-space(.),'Login')] |
  | admin menu    | //nav[contains(@class,'admin-menu')]      |
```

Recipe 312 Loading Selectors from a JSON File

Problem

The same `.hero-banner` selector is now pasted into fourteen scenarios across six feature files. Then the redesign lands and the class becomes `.hero-banner--v2`. It is the Tuesday before release, you are grepping feature files, and you fix eleven of the fourteen, the suite goes red in three places nobody remembers owning. Selectors that are shared in spirit but duplicated in practice always end this way.

Solution

Create `tests/selectors/homepage.json`:

```
{
  "css": {
    "hero":      ".hero-banner",
    "hero cta":  ".hero-banner .btn-primary",
    "nav":       "nav[role='navigation']",
    "footer":    "footer.site-footer"
  },
  "xpath": {
    "page title":  "//h1[contains(@class,'page-title')]",
    "breadcrumb":  "//nav[@aria-label='breadcrumb']"
  }
}
```

Configure `cucumber.js`:

```
worldParameters: {
  selectors: {
    filesPath: './tests/selectors/',
    files: ['homepage.json'],
  }
}
```

Every scenario auto-loads the file at start. Reference names directly:

```
Scenario: Homepage smoke
  Given I am on homepage
  Then I see visible hero, nav, footer
  And I see hero above footer
```

Load a file inline

If you need a file only for one scenario:

```
When I add selectors from "admin-panel.json" file
Then I see visible admin menu
```

The file path is relative to `worldParameters.selectors.filesPath`.

Layering rule

Load order: worldParameters defaults → files listed in worldParameters → inline `add` / `define` steps. Later entries override earlier ones. You can load multiple files to share common selectors across areas.

Under the hood this is a plain merge. The `Before` hook copies the `css` and `xpath` maps from `worldParameters`, then reads each file in `files` from `filesPath` in the order listed and merges its `"css"` and `"xpath"` keys over the top. Both keys are optional, a file can be CSS-only. Two guardrails worth knowing: only `.json` files are accepted (anything else throws a clear error instead of half-loading), and a file that fails to parse fails the scenario immediately, with the file path in the message. You find out at the first scenario, not at assertion forty.

At scale: this file is where multi-project teams win. Keep one `shared.json` per design system, not per project, and distribute it (a git submodule, an npm package, or a plain CI copy step all work). Twenty projects built on the same theme then share one set of names: `hero` means the

same thing in every suite, failure messages read identically across projects, and a markup change in the design system is one pull request to one file instead of twenty. The selector file becomes the contract between QA and front-end, front-end edits the values, QA owns the names.

Recipe 313 Asserting Visibility with Named Selectors

Problem

Post-deploy smoke check, Friday at 5pm. The hero rendered but the CTA inside it did not, and your assertion stops at the first missing element. You fix that, rerun, and discover the headline is missing too. Every discovery costs a full CI round trip. You want one step that checks the whole section and tells you everything that is broken, the first time.

Solution

```
Scenario: Hero section fully rendered
  Given I am on homepage
  And I define css selectors:
    | hero     | .hero-banner       |
    | headline| .hero-banner h1    |
    | cta      | .hero-banner .btn  |
  Then I see visible hero, headline, cta
```

Each name resolves through the registry, then the step runs Playwright's native auto-retrying `locator.waitFor({ state: 'visible' })`, up to 5 seconds per selector. That is polling, not a snapshot: a component that hydrates late still passes, with no static sleep anywhere. And by the time a `Then` step runs, webship-js's AfterStep hook has already run `smartSettle`, so in-flight fetches and pending timers from the previous step are drained, you are asserting against a quiet page. If any name fails, the step keeps going, collects every failure, and throws one error listing all of them. You see everything broken, not just the first.

At scale: a one-line visibility contract is the cheapest smoke test you can run across a portfolio. Give every project the same five names from the shared selector file, tag the scenario `@smoke`, and the nightly run answers "did every site render its critical sections?" in one pass, with failure messages that read identically from project to project. That sameness is what makes a cross-project dashboard worth looking at.

Negation

```
Then I don't see modal, overlay, cookie banner
```

Uses `waitFor({ state: 'hidden' })`. In Playwright terms, "hidden" also covers "not in the DOM at all", so the same step passes for a modal that closed and for one that never rendered. Good for checking a closed modal stays closed.

Recipe 314 Spatial Relation Assertions

Problem

A front-end refactor swaps the header from flexbox to grid. Every element still exists, every visibility check passes, and the screenshot diff is just noisy enough that someone approves it. On the demo call, the nav is sitting under the logo. Presence checks cannot catch this, nothing disappeared. You need assertions about *where things are*: logo left of nav, footer below main, modal on top of the page.

Solution

```
Scenario: Layout contract
  Given I am on homepage
  And I define css selectors:
    | header  | header                |
    | logo    | header .site-logo     |
    | nav     | header nav            |
    | main    | main                  |
    | footer  | footer                |
    | modal   | .modal.is-open        |
  Then I see logo to the left of nav
  And I see header above main
  And I see main above footer
  And I see nav inside of header
  And I see modal over main
```

Each assertion measures real geometry. The step scrolls the element into view if needed (`scrollIntoViewIfNeeded`, with a short grace period), reads its `boundingBox()`, then adds `window.scrollX` / `window.scrollY`, because `boundingBox()` returns viewport coordinates, and you want absolute page coordinates that stay comparable even when two elements are a full screen apart. For `over` / `not over` it also reads the computed `z-index` through `evaluate()`, since Playwright has no native API for computed styles. If either element has no box at all (hidden or detached), the step fails with a message naming which one.

Full list of spatial relations

RELATION	PASSES WHEN
`A above B`	B.top >= A.top + A.height
`A below B`	A.top >= B.top + B.height
`A to the left of B`	A.right <= B.left
`A to the right of B`	A.left >= B.right
`A inside of B`	A's box fully contained in B's box
`A outside of B`	A's box fully contains B's box
`A over B`	A and B overlap AND A has lower (or equal) z-index than B
`A not over B`	A and B do not overlap OR z-indexes do not satisfy `over`

Multi-subject dispatch

```
Then I see heading, subline above list, footer
```

This runs the Cartesian product: `heading above list`, `heading above footer`, `subline above list`, `subline above footer`. All four must pass. Like the visibility steps, the dispatcher checks every pair before failing and reports all broken pairs in one error, four assertions, one CI round trip.

Why spatial assertions matter

Visual regression catches pixel drift. Spatial assertions catch structural drift. A designer swaps `display: flex` for `display: grid` and the nav ends up below the logo instead of beside it. A pixel diff might still match (the elements look similar). A `logo to the left of nav` assertion fails instantly.

At scale: when twenty sites share one base theme, write the layout contract once, header above main, logo left of nav, footer last, against shared selector names, and run it on every site. A theme regression then fails everywhere at once with the same message, which tells you immediately it is the theme and not one project's override. That distinction is exactly what a delivery manager needs the morning a "layout is broken" ticket lands: one red site means a local change, twenty red sites means roll back the theme.

Recipe 315 Viewport Breakpoints by Name

Problem

A "broken on mobile" bug report lands, and nobody can answer the obvious question: which widths do our tests actually cover? One suite hard-codes 375 x 667, another picked 414 x 896, a third has a typo, 786 instead of 768, that has been silently testing the wrong breakpoint for months. Hand-typed viewport numbers drift, and drifted numbers are worse than none, because they look like coverage.

Solution

Webship-js ships named breakpoints out of the box:

NAME	WIDTH	HEIGHT	TARGET
xs	375	667	phone portrait
sm	576	800	large phone / phablet
md	768	1024	tablet portrait
lg	992	768	small laptop / tablet landscape
xl	1200	900	desktop (default)
xxl	1400	900	wide desktop / HD
xxxl	1920	1080	Full HD / large monitor

Use them in Gherkin:

```
Scenario: Mobile layout
  Given I am viewing the site on a xs screen
  And I am on homepage
  Then I see visible hero
  And I see nav to the right of logo

Scenario: Desktop layout
  Given I am viewing the site on a xl screen
  And I am on homepage
  Then I see nav to the right of logo
  And I see cta inside of hero
```

Quoted form for CI clarity

```
Given I am viewing the site on a "xxl" screen
```

Both work. Quote when the name could be mistaken for a word. The step also accepts `device` in place of `screen`, `viewing the site on a "md" device` reads better in some scenarios.

Custom breakpoints

```
worldParameters: {
  selectors: {
    breakpoints: {
      phone:  { width: 360, height: 640  },
      tablet: { width: 900, height: 1280 },
      hd:     { width: 1920, height: 1080, default: true },
    }
  }
}
```

Now `Given I am viewing the site on a phone screen` works. Override via env:

```
export WEBSHIP_SELECTORS_BREAKPOINTS='{"phone":
{"width":360,"height":640}}'
```

Two mechanics worth knowing. First, the step works because webship-js ships `viewport: null` in `playwright.config.ts`, no fixed window size, so when the step resolves the name and calls `page.setViewportSize()`, it is the single authority over the window. Ask for a name that is not defined and you get a friendly error pointing at `worldParameters.selectors.breakpoints`, not a silent fallback. Second, the env var **replaces** the whole breakpoint map, it is parsed as one JSON object, not merged. The export above leaves you with only `phone` defined; include every breakpoint your features mention.

At scale: stop letting each project pick its own numbers. Define the breakpoint map once, export `WEBSHIP_SELECTORS_BREAKPOINTS` at the CI level, and every suite in the organization tests identical widths without touching a single `cucumber.js`. The payoff is an answer your QA lead

can give in one sentence: “every project is tested at 375, 768, and 1200.” Device-lab debates end, and a “broken on mobile” report maps to a breakpoint name the whole team shares.

Recipe 316 Clicking by Named Selector

Problem

The design system replaced your text CTA with an icon button. There is no visible label for `When I click "Login"` to match, the accessible name is an a11y backlog ticket that will not ship this sprint, and the test still has to click the thing today. Pasting the CSS into the step works, and produces exactly the unreadable Gherkin this chapter exists to prevent.

Solution

```
Scenario: Click CTA
  Given I am on homepage
  And I define css selectors:
    | cta | .hero .btn-primary |
  When I click cta
```

The `click <name>` step resolves the name from the registry (CSS map first, XPath second), then calls `locator.click()`, which buys you Playwright’s full actionability pipeline: auto-scroll into view, visible, stable, enabled, not covered by another element. After the click, webship-js’s AfterStep hook runs `smartSettle(page, 1500)`, draining in-flight fetch/XHR calls and pending timers before your next step executes, so whatever the click kicked off has landed before you assert on it. No wrappers, no manual waits.

Click multiple names

```
When I click cta, secondary link
```

Runs each in sequence. If any fails, the step collects errors and reports them together.

Combined with XPath registration

```
When I add "admin link" selector for "//a[contains(normalize-space(.),'Admin')]" xpath selector
And I click admin link
```

Gotcha

The step pattern is liberal: `When I click <anything>`. If the name you give does not exist in the registry, you get an error telling you to define it. It will not fall back to Playwright's text matching, for that, use the classic `When I click "Login"` step.

Recipe 317 Focus and Text Selection

Problem

A screen-reader user reports that focus lands in the wrong field after the page loads, and support cannot reproduce it, their screenshots never show where focus is. The accessibility audit is due Thursday. You need tests that drive keyboard-first flows and pin down the exact state: this field focused, these characters selected, the way a power user or an assistive technology actually experiences the form.

Solution

```
Scenario: Keyboard input
  Given I am on "/register"
  When I move focus to "Email" field
  Then I see Email has focus

  When I fill in "Body" with "Release 2.0 notes"
  And I select from 0 to 7 text in "Body" field

  When I fill in "Title" with "original"
  And I select "original" text in "Title" field
```

Steps:

- `move focus to "<field>" field`, `focus()` on the field (by label, placeholder, name, or id).
- `<name> has focus`, asserts the element is `document.activeElement`.
- `select all text in "<field>" field`, `selectText()` on the field.
- `select from <N> to <M> text in "<field>" field`, `setSelectionRange(N, M)`.
- `select "<text>" text in "<field>" field`, finds the substring, ranges the selection.

Why this matters

Accessibility teams care about keyboard order. Power users care about tab flow. Customer-support screenshots often happen with a field focused, these steps let you reproduce the exact state.

Field resolution order

The steps try, in order:

1. `getByLabel(field)`
2. `getByPlaceholder(field)`
3. `getByRole('textbox', { name: field })`
4. `[name="field"]` or `#field`

First match wins. Accessibility-first lookup, matching Playwright's recommended pattern.

One implementation detail makes this more reliable than it sounds: the chain is built with Playwright's `.or()` combinator into a *single* locator, not four sequential lookups. The whole chain auto-retries together, a label that appears 800ms after navigation still resolves, because the locator keeps polling instead of failing on the first pass. The focus assertion is equally patient: `has focus` waits for the element to attach, then polls `document.activeElement` for up to 5 seconds, so focus moved by JavaScript shortly after load still passes without a sleep. The range-selection steps run `setSelectionRange()` inside the element itself, and the substring variant reads the field's current value first, if the text is not there, you get an error naming the missing substring, not a silent no-op.

Recipe 318 Debugging the Selector Registry

Problem

The new hire's first PR fails with "unknown selector 'admin menu'". They have three JSON files, a `worldParameters` block, and two inline `define` tables in front of them, and no idea which layer was supposed to provide the name, or which layer overwrote it. Guessing by reading config files is slow. You want to see what the scenario actually has registered, at the moment it runs.

Solution

```
Scenario: Debug
  Given I am on "/admin"
  When I add selectors from "admin-selectors.json" file
  Then I print css selectors
  And I print xpath selectors
```

Console output:

```
--- CSS Selectors ---
  "admin menu": nav.admin-menu
  "admin user": .admin-user-link

--- XPath Selectors ---
  "first admin link": //nav[@class='admin-menu']//a[1]
```

When to use it

The print steps dump the live per-scenario registry, the post-merge truth after the `Before` hook has layered `worldParameters`, then each file in `files` in listed order, then every runtime `add` / `define` step executed so far. Place the print *after* the loading step you suspect, and read the output against that order: if the value is wrong, the last layer that defines the name is your culprit; if the name is missing entirely, no layer ever defined it, check the file name and the `filesPath` it resolves against. Once per problem. Remove the print steps before committing, they clutter output.

At scale: put “print the registry, read the merge order” in your triage runbook. When one team maintains suites for many projects, the senior engineer cannot personally debug every unknown-selector failure, a runbook line plus this step turns the same investigation into a five-minute self-serve task on any project.

Recipe 319 Layering Strategy for a Large Project

Problem

One hundred selectors, five teams, one shared file, and on Friday afternoon someone renames `site header` to unblock their own test. Forty scenarios across two other products go red, and the weekend on-call gets paged for a “regression” that is really a naming dispute. Without a layering convention, the selector registry becomes a commons that every team grazes and nobody tends.

Solution

Pick a layering convention:

Layer 1: Shared project selectors, committed in `tests/selectors/shared.json`. Every feature gets them via `worldParameters.selectors.files`.

```
{
  "css": {
    "site header":  "header.site-header",
    "primary nav":  "nav[role='navigation']",
    "site footer":  "footer.site-footer"
  }
}
```

Layer 2: Feature-area selectors, loaded per area.

```
Feature: Admin dashboard
  Background:
    When I add selectors from "admin.json" file
```

Layer 3: Scenario overrides, inline when something is unique to one test.

```
Scenario: Cart totals
  When I add "cart total" selector for ".cart-summary .total-amount" css
selector
  Then I see visible cart total
```

Benefits

- Shared layer, central truth, edited rarely.
- Area layer, owned by feature teams, keeps blast radius small.
- Scenario layer, scratch work that lives with the test.

Name collisions are resolved by last-write: scenario beats file beats worldParameters. That is the intended order. It is the merge mechanics from Recipe 312 doing governance work, the most local intent always wins, and an override is visible right there in the scenario that needs it, instead of hidden in a config file three directories away.

At scale: treat the shared layer like an API. Names are the interface; CSS values are the implementation. Front-end may change values freely, but renaming or deleting a *name* requires a deprecation window where the old name keeps working (two names, one selector) and a heads-up to every consuming team. Version the shared file and distribute it like any dependency, so a project can pin it during a release freeze. This is the pattern we converged on while maintaining suites across dozens of Varbase sites at Vardot: the day selector names became a reviewed contract instead of a free-for-all was the day cross-project breakage stopped being a weekly event.

Recipe 320 Selectors with Responsive Tests

Problem

Mobile regressions keep reaching users because the suite only ever tests the desktop width. The team's first fix was copy-paste: three copies of the homepage scenario with different viewport numbers. Within a month the copies drifted, someone updated the desktop one and forgot the others, and the `xs` copy quietly stopped being maintained. You want one scenario that runs at every breakpoint, so the coverage cannot drift.

Solution

```
Feature: Responsive homepage

  Background:
    Given I define css selectors:
      | header  | header              |
      | logo    | header .logo        |
      | nav     | header nav          |
      | hero    | .hero-section       |
      | cta     | .hero-section .btn  |

  Scenario Outline: Layout at <bp>
    Given I am viewing the site on a <bp> screen
    And I am on homepage
    Then I see visible header, logo, nav, hero, cta
    And I see logo to the left of nav
    And I see header above hero

    Examples:
      | bp  |
      | xs  |
      | md  |
      | xl  |
      | xxl |
```

One feature, four breakpoints. Each `Examples` row runs as an independent scenario: a fresh selector registry from the `Before` hook, a `setViewportSize()` call to the named breakpoint, and spatial geometry measured anew at that width, so `logo to the left of nav` is genuinely re-verified at 375px, not assumed from the desktop run. The scenario name carries the breakpoint (`Layout at xs`), so the report tells you exactly which width broke before you read a single stack trace.

At scale: do the matrix math before you copy this pattern everywhere. Four breakpoints times thirty scenarios times twenty projects is 2,400 scenario runs a night, real CI minutes, real money. The pattern that holds up: run `xs` and `xl` on every pull request, and the full breakpoint matrix nightly under a `@responsive` tag. Then pass `--metadata Breakpoints=xs,md,xl,xxl` to the report generator so the cross-project dashboard shows what each run actually covered. That metadata is the difference between “tests passed” and “mobile is covered”, which is the sentence your delivery manager actually needs.

What this catches

A designer hides the nav on mobile. Your `logo to the left of nav` assertion fails on `xs`. You get a precise error: “nav is not visible at 375 x 667”. You fix the mobile layout, not the test.

Putting It All Together

```
@smoke @layout
Feature: Complete homepage contract

  Background:
    When I add selectors from "homepage.json" file
    And I define css selectors:
      | cookie banner | #cookie-consent |

  Scenario Outline: <bp> viewport renders all sections correctly
    Given I am viewing the site on a <bp> screen
    And I am on homepage
    When I wait page is loaded

    # Visibility
    Then I see visible site header, primary nav, hero, cta, site footer
    And I don't see cookie banner

    # Layout
    And I see site header above hero
    And I see hero above site footer
    And I see primary nav inside of site header

    # Interactive
    When I click cta
    Then I should be on "/get-started"

    Examples:
      | bp  |
      | xs  |
      | md  |
      | xl  |
```

Five named selectors. Three breakpoints. Eight assertions. Zero raw CSS in the Gherkin. Hand this feature to a product owner and they can read every line. Hand it to a new QA engineer and they can extend it without learning your DOM first.

The Philosophy

Named selectors turn tests into documentation. When you write `I see logo to the left of nav`, a product manager with no CSS knowledge understands the intent. When you write `.main-nav__container > ul > li:nth-child(3) a`, nobody does.

Webship-js ~2.0 adds this layer because long-running test suites always start to look like DOM archaeology. Registering selectors once, by name, restores readability without sacrificing precision.

And on teams that maintain many projects at once, names do one more job: they make suites comparable. The same `hero`, the same `primary nav`, the same layout contract, twenty sites, one vocabulary. That is what turns a pile of test suites into a quality signal someone can actually read across a portfolio.

Chapter Summary

- `webship-selectors.js` ships in webship-js ~2.0 and is loaded by default.
- Register named selectors inline (`When I add "..." selector for "..." css selector`), in bulk (`Given I define css selectors:`), or from JSON files (`When I add selectors from "file.json" file`). Files use the `{"css": {...}, "xpath": {...}}` shape and live in `tests/selectors/` by default.
- Spatial assertions: `above` , `below` , `left of` , `right of` , `inside of` , `outside of` , `over` , `not over` .
- Built-in breakpoints: `xs` , `sm` , `md` , `lg` , `xl` , `xxl` , `xxxl` (`xl` is the default). Override via `worldParameters.selectors.breakpoints` or `WEBSHIP_SELECTORS_BREAKPOINTS` , remember the env JSON replaces the whole map. Scroll offset: 60px default, via `WEBSHIP_SELECTORS_OFFSET` .
- `I see <name> has focus` , `I move focus to "<field>" field` , and text-range selection steps drive keyboard-first flows.
- Click any named selector with `When I click <name>` .
- Layer your registry: shared file -> feature area -> scenario overrides. Last write wins.
- Multi-subject dispatch handles Cartesian assertions in one line.
- Debug the active registry with `I print css selectors` / `I print xpath selectors` .

Part IX: Structured Prompt-Driven Development

Chapter 35: Structured Prompt-Driven Development with Webship-js

Concepts and field notes by Shaimaa Natshah, from the Webship.co blog.

Here is the thing. AI changed how we write code. It did not change how we write specifications. SPDD, Structured Prompt-Driven Development, is the discipline that makes that gap visible and closes it. Coined by Thoughtworks and championed by Martin Fowler, SPDD treats prompts as first-class delivery artifacts: version-controlled, reviewed, reused, and improved over time.

Webship-js fits this discipline naturally. Gherkin scenarios are already structured prompts in disguise. They describe what the system should do, in domain language, with executable tests behind them. Combine SPDD with webship-js, and you get a closed loop: human writes the structured prompt, AI generates code, webship-js verifies behavior, and, when reality diverges, you fix the prompt first, then regenerate the code.

This chapter shows the full loop.

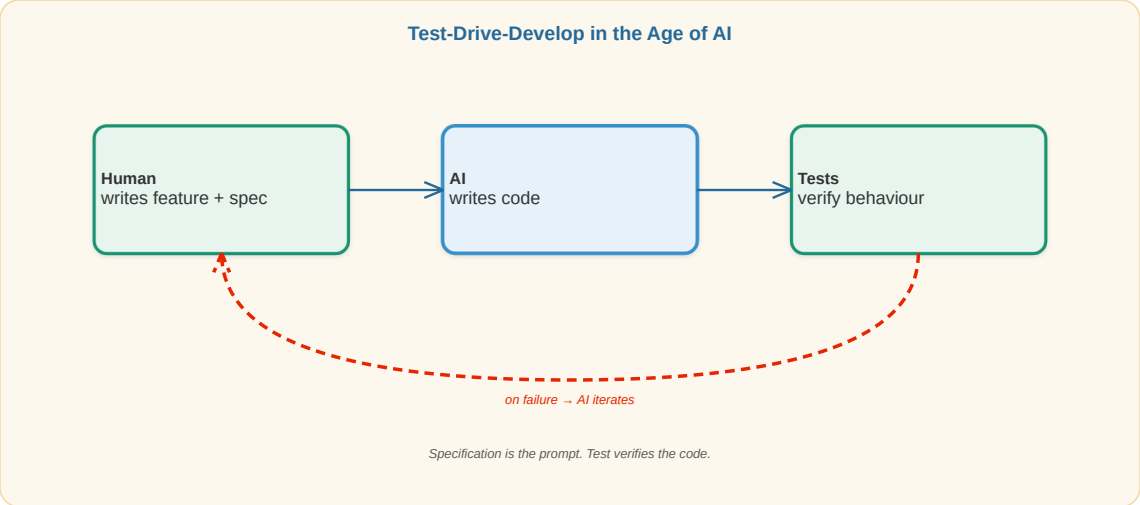


FIGURE 35.1 *Structured Prompt-Driven Development with Webship-js*

What SPDD Actually Means

WHY THIS MATTERS

Vibe coding, chatting your way to a feature, works for prototypes. It collapses for production. SPDD restores engineering discipline to prompt-led work.

Thoughtworks coined SPDD after watching teams use LLM assistants in their internal IT organization. The pattern they saw: ad-hoc chats produce code that ships, then no one can recreate the conversation that produced it. Six weeks later the code needs a small change. The team prompts the model again. The model produces a different design. Drift starts.

Now multiply that by a portfolio. Ten products, each with its own undocumented chat history, is an organization whose most important design decisions are written down nowhere at all.

SPDD fixes this by **versioning the prompt itself**, not just the code. The prompt becomes the source of truth. When code changes, the prompt updates. When reality diverges from the prompt, you fix the prompt first.

Three ideas drive the practice:

1. **Prompts are artifacts.** Commit them. Review them. Reuse them.
2. **Structure beats freeform.** A canvas with named sections forces clarity that "write me a login screen" never will.
3. **Closed loop, not one-shot.** Generate -> review -> refine -> regenerate. Never a single ask-and-ship.

The REASONS Canvas

FIGURE 35.2 *The REASONS Canvas*

The seven sections move from abstract intent ("why are we doing this") to concrete execution ("what must we not do"). The first four are abstract and design-focused. **Operations** is the bridge, it pins down concrete signatures and steps. The last two govern: **Norms** for engineering standards, **Safeguards** for invariants the system must never violate.

The order is also a review router. Product owners can sign off on Requirements and Entities without reading code. Architects argue about Approach and Structure. QA owns Operations. Security gets a named place to put non-negotiables instead of a comment thread that scrolls away. One canvas, four audiences, no translation step.

> **NOTE**
>
> When the LLM generates code from a complete REASONS canvas, the output stays inside the boundary you defined. Hallucinations drop. Reviews speed up.

How Webship-js Maps to REASONS

Webship-js artifacts already cover most of the canvas. Here is the mapping:

REASONS SECTION	WEBSHIP-JS ARTIFACT
Requirements	`Feature:` description (As / I want / So that)
Entities	Domain nouns referenced in step text
Approach	Background section + scenario outlines
Structure	`tests/` layout + `cucumber.js` `require:` + custom step files
Operations	Individual `Given / When / Then` steps
Norms	`worldParameters` + project conventions
Safeguards	Tags (`@critical`, `@security`), assertions, `Then` invariants

A well-written `.feature` file is a structured prompt. The novelty SPDD adds is the **discipline** of treating it like one, versioning it, reviewing it, never letting it drift from the code it specifies.

Two rows deserve a closer look. Norms map to `worldParameters` because that is where project-wide policy already lives in webship-js, the `launchUrl` each environment overrides with the `LAUNCH_URL` env var, the selector registry every scenario inherits, the breakpoint map your responsive scenarios share. And Safeguards map to tags because tags are *executable* policy: `@critical` decides what blocks a merge, and a `@js-fail` tag turns any JavaScript console error into a hard failure for that scenario. A safeguard written as a tag gets re-checked on every run. A safeguard written in a wiki gets re-checked never.

Recipe 336 Writing a Structured Prompt as a Webship-js Feature

Problem

Sprint planning. The product owner drops “users should be able to reset their password” into the backlog and moves on. QA writes scenarios from guesswork, the developer makes different guesses in code, and three weeks later security asks why there is no rate limit, a requirement that existed only in someone’s head. The story was never wrong; it was never *structured*. You want to turn it into a prompt precise enough that an AI, or a new teammate, can build from it, and webship-js can verify it.

Solution

Apply the REASONS canvas inside the feature file:

```
Feature: Password reset

  # R - Requirements
  # As a registered user
  # I want to reset my password via email
  # So that I can regain access if I forget it
  # Definition of done:
  #   - email arrives within 60 seconds
  #   - token expires after 24 hours
  #   - reset succeeds; old password no longer works

  # E - Entities
  # User (email, password_hash, reset_token, reset_token_expires_at)
  # ResetEmail (to, subject, body, token_link)

  # A - Approach
  # Use a single-use signed token. Email it. Validate on submit.
```

```
# S - Structure
# Pages: /forgot-password, /reset-password?token=...
# API: POST /api/auth/forgot, POST /api/auth/reset

# O - Operations (these become the scenarios below)
# N - Norms
# All copy in English + Arabic. WCAG AA. Rate-limited 3/hour.
# S - Safeguards
# Token never logged. Rate limit enforced. Old session invalidated.

Background:
  Given I am on "/forgot-password"

@critical @auth
Scenario: Request reset email
  When I fill in "Email" with "user@example.com"
  And I press "Send reset link"
  And I wait for AJAX to finish
  Then I should see "Check your inbox"
  And the response status code should be 200

@critical @auth @security
Scenario: Token expires after 24 hours
  Given a reset token was issued 25 hours ago for "user@example.com"
  When I am on "/reset-password?token=expired-token"
  Then I should see "This link has expired"
  And I should not see "New password"

@critical @auth @security
Scenario: Old password stops working after reset
  Given I have a valid reset token for "user@example.com"
  When I am on "/reset-password?token=valid-token"
  And I fill in "New password" with "NewSecure123!"
  And I press "Reset password"
  And I wait for AJAX to finish
  Then I should see "Password updated"
  When I am on "/login"
  And I fill in "Email" with "user@example.com"
```

```
And I fill in "Password" with "OldPassword456"
And I press "Sign in"
Then I should see "Invalid credentials"
```

The comment block at the top is the REASONS canvas. The scenarios are the **Operations**, the concrete, testable, signature-precise steps. Norms (Arabic copy, WCAG AA, rate limit) and Safeguards (token never logged, rate limit, session invalidation) are encoded as additional scenarios or tags that downstream steps verify.

Why this works

The whole document is committed to git. Reviewers see intent ®, domain (E), strategy (A), boundaries (N + S), and verification (Operations) in one place. When an engineer six months later asks "why does the reset flow rate-limit?", the answer is right there in the prompt that drove the implementation.

Look closely at the Operations layer, because it mixes two kinds of steps on purpose. `I fill in "Email" with ...`, `I press ...`, `I should see ...`, and `the response status code should be 200` are built-in webship-js steps, the fill step resolves the field label-first, and the AJAX wait hands control to smartSettle, which watches the page's instrumented fetch/XHR counter and pending timers until both hit zero and the DOM goes quiet. But `Given a reset token was issued 25 hours ago for ...` is a domain step that does not exist yet. That is not a mistake, it is the canvas doing its job. The Operations section pins down the *signature* of the custom step before anyone implements it, so the step definition becomes part of the work the prompt specifies. Notice also that the expired-token scenario asserts the absence of the "New password" field rather than "the form is gone", assert on what a user would see, and the step stays readable and executable.

At scale: adopt one canvas shape across every product and the payoff compounds. A product owner who learns to read the R block on one project can sign off on any project. A security reviewer greps every repo for the `S - Safeguards` comment block and audits the whole portfolio in an afternoon. And acceptance criteria stop being rewritten by QA from scratch, the PO's words land in the R block, the scenarios underneath are visibly derived from them, and that derivation is what makes a green run something a non-engineer can trust as sign-off.

Recipe 337 The SPDD Loop with AI-Generated Code

Problem

You have the canvas. Now the temptation is to paste fragments of it into a chat window, copy code back out, and eyeball the result, which is exactly the lossy, unrepeatable loop SPDD exists to kill. The handoff to AI needs to be as structured as the prompt itself: one committed artifact in, one verified suite out, nothing important living only in a chat transcript.

Solution

The full loop:

```
# 1. Author the structured prompt
git add tests/features/password-reset.feature
git commit -m "spec: password reset (REASONS canvas)"

# 2. Hand it to AI
claude "Read tests/features/password-reset.feature.
        Implement the controllers, model, mailer, and views to make every
        scenario pass. Stay inside the Norms and Safeguards. When you
finish,
        run the suite."

# 3. AI generates code, runs the suite
# 4. On failure, AI iterates
# 5. On success, you commit code AND any prompt refinements
```

The webship-js suite is the verification step. If the suite is green, the AI has produced code that satisfies the structured prompt. If red, the AI iterates, guided by the failing step, not by you re-explaining the spec.

Notice what step 5 says: commit the code *and any prompt refinements*. Generation almost always teaches you something about the spec, an ambiguous Norm, an entity field nobody mentioned. Folding that back into the feature file is the difference between SPDD and a one-shot generation with extra ceremony.

Where webship-js shines

The Cucumber output names the failing scenario, the failing step, and the expected/actual, and the default profile also writes a machine-readable JSON report to `tests/reports/cucumber_report.json`. An AI agent does not need to parse console colors; it reads the JSON, finds the failed step, and patches the code. That same JSON is what `bin/generate-reports.js` turns into the HTML and PDF a human reads, so the artifact the AI iterates against and the artifact a delivery manager signs off on are the same file, one source, two readers. The default 45-second step timeout is a quiet safety net here too: generated code that hangs gets killed and reported as a failure instead of stalling the loop. No human bottleneck.

> **PRO TIP**
>
> Tag the most important scenarios `@critical`. Run `npx cucumber-js --tags "@critical"` first in CI. Fast feedback for the SPDD loop.

At scale: the loop is cheap to run once and expensive to run carelessly across a portfolio. Every generation pass is tokens, so the teams that survive a budget review run the *deterministic* half everywhere and the *generative* half selectively: the `@critical` cucumber-js run gates every PR in every repo, while the “hand it to AI and regenerate” pass is reserved for the long-lived products where regeneration is actually practiced. Wire the same `--tags "@critical"` profile into a shared CI template so “fast feedback” means the same scope on all twenty suites, and tag the JSON report with `--app-version` and `--env` metadata so a green run on product C is comparable to a green run on product K, the difference between a portfolio quality signal and twenty incomparable thumbs-up.

Recipe 338 When Reality Diverges, Fix the Prompt First

Problem

Tuesday morning, a bug report: password reset emails sometimes land in spam. The engineer on triage can patch the mailer config in ten minutes, and every instinct says do exactly that. But the ten-minute fix updates the code and leaves the prompt ignorant, the spec now describes a system that no longer exists, and the next regeneration will faithfully rebuild the bug. Where does the change start?

Solution

Not in the code. SPDD's most important rule:

> **IMPORTANT**
>
> When reality diverges from the prompt, fix the prompt first. Then regenerate or update the code.

Steps:

1. Open `tests/features/password-reset.feature`.
2. Update **Norms** to add: “All transactional email must include valid SPF, DKIM, and DMARC headers.”
3. Add a new scenario verifying the headers:

```
@critical @auth @deliverability
Scenario: Reset email passes SPF / DKIM / DMARC
  When I fill in "Email" with "user@example.com"
  And I press "Send reset link"
  And I wait for AJAX to finish
  Then the API response status code should be 200
  And the most recent email to "user@example.com" should pass SPF
  And the most recent email to "user@example.com" should pass DKIM
  And the most recent email to "user@example.com" should pass DMARC
```

4. Run the suite. The new scenario fails (no implementation yet).
5. Hand the updated prompt to AI, or implement manually.
6. Suite goes green. Commit the prompt change AND the code change in one commit.

Why this matters

A code-first fix would patch the mailer config and forget to update the spec. Six months later, a refactor removes the SPF setup, the spec says nothing about it, and the bug returns. Prompt-first fixes prevent that decay.

Watch the order of operations in step 4: the new scenario must fail *before* anyone implements anything. That red run is evidence the check actually checks something, the same red-first discipline TDD has always used, applied to a prompt. The email-header steps themselves (`should pass SPF` and friends) are custom step definitions your team writes once against a mail-capture service; after that, every deliverability Norm in every product is one Gherkin line away from being enforced.

At scale: prompt-first fixes are greppable, and that is their quiet superpower. When the same spam-folder bug shows up on a second product, you search every repo's feature files for "DKIM" and get an instant map of which products enforce deliverability and which never got the Norm. A delivery manager asking "did the fix from product A reach products B through F?" gets an answer from grep, not from interviewing six teams.

Recipe 339 Three SPDD Skills Mapped to Webship-js

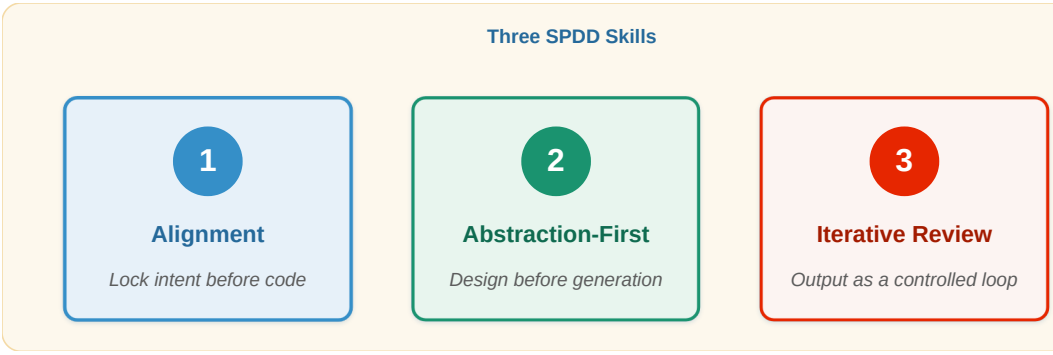


FIGURE 35.3 *Three SPDD Skills Mapped to Webship-js (Recipe 339)*

Thoughtworks names three skills that make SPDD work. Each maps directly to a webship-js practice:

1. Alignment, "Lock intent before you write code"

In webship-js: write the **Feature: … As … I want … So that** narrative *before* any scenarios. This is the alignment moment. Show it to product, design, security. Get the nods. Only then write scenarios. The narrative costs five minutes to change; the forty scenarios underneath it cost a sprint. Arguing at the cheap layer is the whole point, and it is the last moment a product owner can redirect the work by editing three lines of plain English.

2. Abstraction-First, “Design before you generate”

In webship-js: define the **Background** and the **selectors** before writing the per-scenario steps. Decide what entities you are talking about. Decide what page objects or named selectors will exist. Write a “Given I define css selectors:” data table. Now scenarios slot into a clean abstraction. This matters double when AI generates the scenarios: a model handed an existing selector vocabulary reuses it, while a model handed nothing invents a fresh CSS selector per scenario, and you inherit the cleanup.

3. Iterative Review, “Output as a controlled loop”

In webship-js: never let AI run unattended for more than a single feature at a time. After every generation, review:

- Did the new scenarios respect the existing Background?
- Did the AI introduce new selectors when existing ones would have worked?
- Did any Safeguard get bypassed?

Then iterate. Tighten the prompt. Regenerate. Repeat. The review questions above are checkable in a diff, “did any Safeguard get bypassed?” is a question about scenarios and tags, not about reading every generated line of implementation. That is what keeps the loop fast enough to actually run.

At scale: these three skills are worth standardizing as review gates, not leaving to each team’s taste. Write the three review questions into the PR template every repo shares, and “did the AI invent a new selector?” stops depending on whether the reviewer happened to think of it. The selector half of that question gets sharper when you back it with a committed `tests/selectors/` registry per product: a model handed an existing selector vocabulary reuses it, so the bigger the registry the team curates, the less cleanup any single generation produces. When all twenty products run the same alignment-then-abstraction-then-review loop, a new QA lead moving between them inherits one workflow instead of twenty dialects, and a manager auditing “are teams reviewing AI output, or rubber-stamping it?” reads PR histories with the same three questions in every repo.

Recipe 340 SPDD-Style Prompt Templates for AI Agents

Problem

The canvas worked, and then it spread. Three sprints in, five engineers and two AI agents are authoring feature files, and every one of them remembers REASONS slightly differently. One drops Entities. One merges Norms into Safeguards. The agent, asked from a blank context, invents an eighth section. Reviews slow to a crawl because every file has its own shape, and the new hire's first PR copies whichever broken variant they saw first. The structure needs to come from the repo, not from memory.

Solution

Drop a `templates/spdd-feature.md` in your repo:

```
# Feature name

## R - Requirements
- As a [role]
- I want [capability]
- So that [outcome]
- Definition of done: [bullet list]

## E - Entities
- [Noun]: [fields]

## A - Approach
- [One paragraph]

## S - Structure
- Pages / Routes:
- API:
- Components:

## O - Operations
- [List of scenarios to write]

## N - Norms
- i18n:
- Accessibility:
- Performance:
- Logging:

## S - Safeguards
- Security:
- Privacy:
- Rate limits:
- Failure modes:
```

Add to `.claude/CLAUDE.md`:

```
When asked to author a webship-js feature file:
1. Read templates/spdd-feature.md.
2. Fill every section before writing any Gherkin.
3. Place the filled-in REASONS canvas as comments at the top of the
.feature file.
4. Generate one scenario per item in the Operations section.
5. Tag every scenario with the relevant Norm/Safeguard category.
```

Now every AI-driven feature file starts from the same canvas. The split matters: the template is *data* (what a finished canvas looks like) and the CLAUDE.md instruction is *policy* (when and how to use it). An agent given both cannot improvise the structure away. Rule 2, fill every section before writing any Gherkin, forces the same order the canvas teaches humans: alignment first, operations last. An agent that cannot fill in Entities has found a gap in the story, not a formatting problem, and it surfaces that gap before a single scenario exists. Rule 5 wires Norms and Safeguards into tags, which means the governance sections stop being comments and start being things `--tags` expressions can run.

Because the template itself is committed, it is versioned prompt infrastructure. When the security team decides every Safeguards section needs a “data retention” line, that is one PR to one file, and every feature authored after it inherits the change. Compare that to chasing five engineers’ personal prompt stashes.

At scale: the template stops being per-repo furniture and becomes a shared dependency, a small package, or a file your project scaffolder drops into every new repo, so the canvas shape is identical on product 1 and product 21 without anyone copying it by hand. That single point of change is the whole payoff. When governance evolves, the security team opens one PR against the shared template, every project that pulls the update authors its next feature with the new Safeguards line, and a compliance auditor checks one diff instead of interviewing twenty teams about their prompt conventions. The cost the pattern removes is the silent one: twenty slightly different canvases, none of them wrong, none of them comparable, each one a small tax on every review and every onboarding.

PRO TIP

Pair this with the `webship-js-ai-agent` Claude subagent. The agent already knows the step definitions; the template gives it the structure.

Recipe 341 The webship-js Stage Map

A team that has just bought into SPDD always asks the same question in the first week: “fine, but which webship-js thing implements which stage?” The new QA lead, onboarding onto their third product, does not want philosophy, they want to know where intent lives, where scope lives, and what command proves the prompt is satisfied. Without an agreed map, each project answers differently, and the discipline dissolves into local dialects.

The webship-js step library makes Operations easy to express. Here is the stage map most teams converge on:

SPDD STAGE	WEBSHIP-JS MECHANISM
Capture intent	`Feature:` description + `As/I want/So that`
Lay down domain	Comment block before `Background`
Set scope	`@tags` on scenarios (`@critical` , `@auth` , `@a11y`)
Generate operations	`Scenario:` blocks with Given/When/Then steps
Encode norms	`worldParameters` config + locale/a11y scenarios
Encode safeguards	`@security` tags + assertion scenarios + AJAX-wait discipline
Verify	`npm run test:chromium`
Refine prompt	Edit feature file, commit, regenerate code

Two rows do quiet heavy lifting. The verify row is deliberately a single command: `npm run test:chromium` pins the browser through the `BROWSER` env var, so "the prompt is satisfied" means the same thing on a laptop and in CI, no argument about which run counts. And the safeguards row lists "AJAX-wait discipline" alongside tags because webship-js enforces it mechanically: after every state-changing step, an AfterStep hook runs smartSettle, which watches the instrumented fetch/XHR counter, pending timers, and DOM mutations until the page is genuinely quiet. A safeguard scenario that asserts on a half-rendered page would be a safeguard in name only; the settle engine is what makes the assertion mean something.

This is SPDD with webship-js bolted on. The discipline scales to teams.

At scale: the map is the thing you standardize across a portfolio, not the individual feature files. When all twenty projects use the same stage map, a delivery manager can ask "show me the Safeguards for product X" and get the same artifact shape every time, tags plus assertion scenarios, greppable, runnable. Cross-project questions ("which products encode an accessibility Norm?") become one search over `@a11y` instead of twenty status meetings. Print the table, pin it in the team wiki, and treat deviations from it as review findings.

When to Use SPDD with Webship-js

BEST PRACTICE

SPDD pays off when stakes are high and lifetime is long. Skip it for prototypes.

High-value scenarios (use SPDD):

- Regulated systems (finance, healthcare, government).
- Enterprise CMS where business logic accumulates over years.
- Customer-facing flows with security or accessibility requirements.
- Any feature whose behavior must remain stable across releases.

Low-value scenarios (skip SPDD):

- Throwaway prototypes.
- One-off internal scripts.
- UI explorations where you have not yet decided what the feature does.

The cost of SPDD is the time to fill the canvas. The benefit is decades of preserved intent. Pick the projects where the benefit clears the cost.

SPDD vs Spec-Driven Development

You will see related terms. The differences matter:

PRACTICE	WHAT IT ASKS
Vibe coding	Chat with AI until something works.
Spec-driven development (SDD)	Write specs as Markdown, hand to AI, generate code.
Structured prompt-driven (SPDD)	Write specs in a fixed canvas (REASONS), version them, regenerate when prompts change. Code is regenerated; the prompt is the source of truth.
Test-Drive-Develop (this book)	Write executable specs (Gherkin), AI generates code, tests verify. Prompt + tests are joint sources of truth.

Webship-js sits at the intersection of SPDD and Test-Drive-Develop. The Gherkin file *is* the structured prompt. The step definitions *are* the executable verification. The pair is unbreakable, if the test passes, the code matches the spec by definition.

Recipe 342 SPDD-Driven CI Gate

Problem

The discipline holds for exactly as long as every engineer holds it. Then comes the Friday-afternoon hotfix: code patched, suite green, prompt untouched, and nobody notices, because nothing *checks*. Six weeks later someone regenerates from the feature file and faithfully rebuilds the pre-hotfix behavior. Fix-the-prompt-first cannot live on goodwill alone; it needs a gate that fails the merge when prompt and code have quietly parted ways.

Solution

Add a CI step that:

1. Re-runs the AI generation in dry-run mode against the current `.feature` files.
2. Diffs the output against the committed code.
3. Fails the job if the diff is non-empty.

```yaml
# .github/workflows/spdd-check.yml
name: SPDD prompt-code sync

on: [pull_request]

jobs:
  spdd-sync:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Run SPDD generator (dry-run)
        run: |
          npx claude-code --headless \
            --prompt "Read every tests/features/*.feature file. Confirm
the implementation in src/ matches every scenario. List any divergences."

      - name: Run webship-js suite
        run: npx cucumber-js --tags "@critical"

      - name: Comment PR with SPDD report
        if: failure()
        run: gh pr comment ${{ github.event.pull_request.number }} --body
"SPDD divergence detected. Fix the prompt first, then regenerate code."
```

The gate has two halves, and they fail differently. The cucumber-js run is deterministic: `--tags "@critical"` executes only the scenarios you marked as merge-blocking, each step gets the default 45-second timeout, and a failure means the code demonstrably does not do what the prompt says. The AI divergence check is probabilistic: it asks the model to *read* rather than write, compare every scenario against `src/` and list mismatches, which catches drift the suite cannot, like a Norm ("rate-limited 3/hour") that no scenario asserts yet. Trust them accordingly. The deterministic half can block merges from day one; the probabilistic half should run nightly first, until you have seen its false-positive rate with your own eyes. The PR comment matters more than it looks: it tells the engineer *what to do*, fix the prompt first, at the exact moment they are about to do the opposite.

This is heavier than a normal CI gate. Run it nightly in early adoption, on every PR once the team trusts it.

At scale: the LLM half of this gate has a per-run token cost, and across forty repositories on every PR that becomes a real line item. The pattern that survives the finance review: deterministic `@critical` gate on every PR in every repo (cheap, fast, zero false positives), prompt-sync check nightly and only on the products where regeneration is actually practiced, usually the regulated, long-lived ones from the "When to Use SPDD" list. Publish the workflow as a shared template so the gate itself cannot drift project by project, which would be a faintly ridiculous way for an anti-drift gate to fail.

The SPDD Mindset

The most surprising thing about SPDD is how little it changes about the code. You still write code. You still test. You still ship. What changes is **what you treat as primary**.

Before SPDD: code is primary, specs are documentation. After SPDD: prompts are primary, code is the artifact the prompt produces.

This is not academic. It changes how you debug. It changes how you onboard. It changes how AI-driven refactors stay safe over years.

Webship-js was built for the same world. Every step definition is a verb. Every feature file is a contract. Stack SPDD on top, and you have a system where:

- Prompts are committed.
- Tests are executable.
- AI generates code.
- Tests verify.
- Failures point at the prompt, not at the engineer's coffee intake.

That is the discipline. That is the loop. Master it.

Chapter Summary

- SPDD treats prompts as first-class delivery artifacts: version-controlled, reviewed, reused.
- The **REASONS canvas** structures every prompt: Requirements, Entities, Approach, Structure, Operations, Norms, Safeguards.
- Webship-js feature files are structured prompts in disguise, the REASONS canvas maps cleanly to feature comments + Gherkin.
- Three required skills: **Alignment** (lock intent first), **Abstraction-First** (design before generation), **Iterative Review** (controlled loops, not one-shot prompts).
- The cardinal rule: **when reality diverges, fix the prompt first; then update the code.**
- High-value SPDD scenarios: regulated, long-lived, complex business logic. Skip it for prototypes.
- Webship-js sits naturally at the SPDD + Test-Drive-Develop intersection, Gherkin is the prompt, step definitions are the verification.
- SPDD-style CI gates can re-verify prompt-code sync on every PR.

Part X: Modern Test Authoring

Chapter 36: Web-First Assertions

In older test frameworks, the rhythm was always the same: wait for AJAX, then assert. If you skipped the wait, the assertion fired against stale DOM and your test failed. Webship-js ~2.0 ships a family of **web-first assertions** that fold the wait into the assertion itself. You write a single line. It polls the page every 100ms until the matcher passes or the time budget runs out, 5 seconds by default, or whatever you set with `within N seconds`.

The assertion IS the wait. No more "wait for AJAX, then check." No more arbitrary `I wait 3 seconds`. And here is the part teams keep missing: the budget is a cap, not a sleep. A passing assertion returns on the first successful poll. A 10-second budget on a healthy page costs you about 100 milliseconds.

This chapter is built around the new web-first family added in the latest webship-js release: state checks (`visible`, `hidden`, `focused`, `enabled`, `disabled`, `editable`, `attached`), value/text/attribute/class/count assertions, role-aware visibility, and viewport-membership checks, each with an optional `within N seconds` budget override.

Why Web-First Assertions Matter

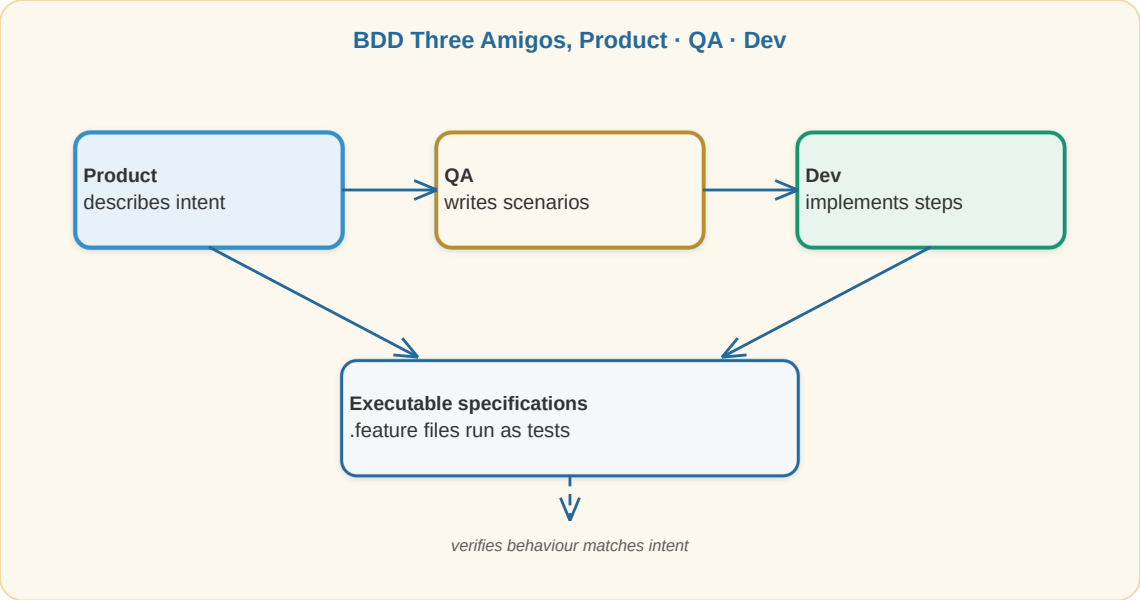


FIGURE 36.1 *Why Web-First Assertions Matter*

A traditional Selenium-era test reads like a transcript of computer fights:

```
1. Click button
2. Wait 2 seconds (hope for the best)
3. Check if the success message is there
4. If not, sleep 2 more seconds and check again
```

A web-first test reads like a behaviour description:

```
When I click "Submit"
Then ".success-banner" should be visible within 10 seconds
```

Same intent. Half the steps. Zero flakiness from arbitrary sleeps.

There is a second layer working underneath, and knowing it explains why these assertions feel so calm. After every state-changing step, webship-js runs `smartSettle(page, 1500)`, it watches `window.__webshipAjaxCount` (every fetch and XHR in flight), `window.__webshipPendingTimers` (queued `setTimeout` callbacks), and `window.__webshipLastMutation` (the timestamp of the last DOM change), and waits until the page has been quiet for at least 250ms. So by the time your `Then` runs, most of the storm has already passed. The web-first assertion covers the one thing settle cannot know: whether *your* element reached *your* expected state. Settle stabilises; the assertion verifies. You need both, and you get both without writing either.

Recipe 343 Asserting Element Visibility With a Budget

Problem

Your contact form posts to a backend that answers in 200ms on a quiet Tuesday morning and in 8 seconds during the load spike before a release. The test you wrote on your laptop passes every time; the same test fails twice a week in the nightly run, and every triage ends with “it’s fine, just re-run it.” The old fix was `I wait 10 seconds`, which made every run slow to protect against the worst case.

Solution

```
Feature: Submit form

  Scenario: Successful submission shows banner
    Given I am on "/contact"
    When I fill in "name" with "Maria"
    And I fill in "email" with "maria@example.com"
    And I press "Send"
    Then ".success-banner" should be visible within 10 seconds
```

The poll loop checks visibility every 100ms until the banner appears or the 10-second deadline passes. Default budget without `within` is 5 seconds. The first successful poll wins immediately, on a fast backend this step finishes in a few hundred milliseconds even with a 10-second budget. And most of the waiting has often already happened before the assertion runs: the AfterStep hook ran `smartSettle(page, 1500)` after the click, watching `window.__webshipAjaxCount` drop back to zero. The assertion covers what settle cannot, the render that happens *after* the response lands. One ceiling to respect: the Cucumber step timeout is 45000ms, so keep `within` budgets comfortably under 45 seconds.

At scale: standardise budgets instead of letting each author guess. A sensible portfolio rule: default 5 seconds everywhere, plus a short, documented exception list for known-slow endpoints. Generous budgets cost nothing on green runs, polling exits on first pass, they only delay the discovery of true failures, which is the right trade. What hurts across 20 projects is twenty different opinions about what "slow" means, sprinkled as random `within` clauses nobody can review.

Recipe 344 Asserting Element Is Hidden

Problem

Your settings modal closes with a 300ms fade-out. The element stays in the DOM the whole time, the framework keeps it mounted and toggles CSS. A test that checks for absence right after Escape fails randomly on slower CI machines because the animation has not finished, and passes on your laptop because it has. Classic "works on my machine," except this time the machine works and the test does not.

Solution

```
Scenario: Modal closes on Escape
  Given I am on "/dashboard"
  When I click "Open Settings"
  And I press the key "Escape"
  Then ".modal" should be hidden within 2 seconds
```

`hidden` accepts both “removed from DOM” and “present but not visible”, mirroring what a real user perceives. The matcher polls every 100ms, so the 300ms fade costs three polls and the step passes the moment `display: none` (or detachment) lands. This is also why you assert `hidden` rather than checking the element left the DOM: modern frameworks keep modals mounted for re-use, and a DOM-absence check would fail forever against a perfectly working close button.

Recipe 345 Asserting Element Is Attached

Problem

Your product table renders 2,000 rows through a virtualised list, DOM nodes mount only as the user scrolls near them. The new hire writes `should be visible` for row 50, watches it fail, and spends an hour convinced the data is missing. The data is fine. The row mounts after scrolling but sits outside the viewport, and “visible” means exactly what it says.

Solution

```
Scenario: Row 50 mounts after scroll
  Given I am on "/products"
  When I scroll to bottom
  Then "[data-row='50']" should be attached within 5 seconds
```

`attached` checks DOM presence and nothing else, no CSS visibility, no viewport math. The poll asks “does a node matching this selector exist in the document?” every 100ms. That is exactly the right question for virtualised lists, where mounting is the behaviour under test. Use `visible` when you care about what the user actually sees, `attached` when you care about what the framework actually rendered. Mixing them up is the most common reason a correct page fails a correct-looking test.

Recipe 346 Asserting Focused Element

Problem

After form validation fails, the first invalid field should receive focus so a keyboard user lands exactly where the fix is needed. This contract breaks silently: a refactor swaps the validation library, focus stops moving, and no sighted mouse user ever notices. The bug report arrives weeks later from a screen-reader user, or from the accessibility audit you are paying for.

Solution

```
Scenario: Invalid email re-focuses email field
  Given I am on "/signup"
  When I fill in "email" with "not-an-email"
  And I press "Sign up"
  Then "#email" should be focused within 1 second
```

Focus is a keyboard-accessibility contract. The web-first matcher polls `document.activeElement` every 100ms until it matches your selector, so the assertion tolerates the small delay between the validation handler running and the `.focus()` call landing. The tight 1-second budget is deliberate, focus management is near-synchronous, and a long budget would only slow the discovery of a real regression.

At scale: make a focus assertion part of the definition of done for every form across your products. It is one line per form, it costs nothing on green runs, and it catches the class of regression that otherwise reaches you through support tickets from your least-served users. Pair it with the audit step (`Then the page should pass an accessibility audit at level "AA"`), the audit catches structural issues, the focus assertion catches behavioural ones.

Recipe 347 Asserting Enabled / Disabled

Problem

A “Save” button stays disabled until the form is valid. Two sprints ago someone refactored the validation logic, and on one of your sites the button never enabled again, users filled in everything and stared at a gray button. It took three days and one angry customer to surface, because every manual tester filled the form in the one “right” order that happened to still work.

Solution

```
Scenario: Save unlocks after required fields
  Given I am on "/profile"
  Then "#save" should be disabled
  When I fill in "name" with "Lina"
  And I fill in "email" with "lina@example.com"
  Then "#save" should be enabled within 2 seconds
```

`disabled` and `enabled` are mirror predicates sharing the same 100ms polling logic. They read the element’s real disabled state, the `disabled` property as the browser sees it, not the styling. That distinction matters both ways: a button styled gray but still clickable is a bug this catches, and a button styled normally but dead to clicks is the other bug it catches. Note the first assertion carries no `within` clause: it uses the default 5-second budget and passes on the first poll, because the page arrived settled, navigation already ran through smartSettle before the step executed.

Recipe 348 Asserting Editable

Problem

Profile fields ship read-only until the user clicks “Edit”. After a design-system upgrade, the field *looked* editable, live border, text cursor, the works, but the `readonly` attribute never came off. Users clicked, typed into a void, and assumed they had saved. An “enabled” assertion would have passed the whole time; the field was enabled. You need the stronger check.

Solution

```
Scenario: Edit unlocks the input
  Given I am on "/profile"
  Then "input[name=display-name]" should not be editable
  When I click "Edit"
  Then "input[name=display-name]" should be editable within 2 seconds
```

`editable` is stronger than `enabled`, it checks the `readonly` attribute, the `disabled` attribute, and the contentEditable state in one call, polled on the same 100ms cadence. An input can be enabled yet read-only, and a rich-text `div` can be neither an input nor disabled yet still refuse typing. `editable` answers the only question the user cares about: if I type, do characters appear? The negative form before the click pins the starting state, so the test proves the *transition*, not just wherever the field happened to end up.

Recipe 349 Text-Containment Assertion With Budget

Problem

After clicking “Apply”, a coupon summary updates somewhere in the cart sidebar. Marketing rewrites the surrounding copy every quarter, “You saved”, “Discount applied”, “Your savings”, and every rewrite used to break the old exact-match test. The one thing that never changes is the discount amount itself. Anchor on that.

Solution

```
  Scenario: Coupon discount appears
    Given I am on "/cart"
    When I fill in "code" with "SAVE10"
    And I click "Apply"
    Then ".cart-summary" should contain text "Discount: -$10.00" within 5
seconds
```

The matcher does a substring match against the element’s rendered text, re-read on every 100ms poll, so debounced updates, streamed totals, and slow re-renders are all caught the moment the substring appears. Scope the selector to the tightest stable container you have: a substring asserted against `body` will eventually pass for the wrong reason, an error message carrying the same digits, a cached value elsewhere on the page. For regex or whole-text equality, use the dedicated steps in the next two recipes.

At scale: substring assertions and translated sites do not mix. “Discount” is “Rabatt” on your German brand site and something else entirely on the Arabic one. When the same scenario must run across locales, anchor on locale-stable substrings, amounts, currency codes, SKUs, or resolve elements through your shared selectors JSON in `tests/selectors/`, and keep the human words out of the assertion.

Recipe 350 Exact Text Assertion

Problem

A status badge should read exactly `Active`. The old test used a substring check, and `Inactive` contains `Active`, so when a deploy inverted the status logic, the suite stayed green while every dashboard in production showed the wrong state. One character of slack, one afternoon of incident review.

Solution

```
Scenario: Status badge is exact
  Given I am on "/users/42"
  Then ".status-badge" should have text "Active" within 2 seconds
```

`should have text` does exact, whitespace-trimmed equality on every poll. The trim matters more than it looks: templates pad text nodes with newlines and indentation, and without trimming, `"\n Active\n"` would never equal `"Active"`. Use `should contain text` when you genuinely want substring semantics, and reach for exact equality any time one valid value is a substring of another, which happens constantly with status words (Active/Inactive, Done/Undone, Lock/Unlock).

At scale: status labels are a vocabulary, and vocabularies drift. One product says “Active”, the next says “Enabled”, the third says “On”. Keep a short glossary of canonical status strings per product family and let exact-text assertions enforce it on every run, when a developer “improves” a label, the suite tells you before the translations, the documentation, and the screenshots all start disagreeing with each other.

Recipe 351 Value Assertion on Inputs

Problem

A URL slug field auto-populates from the title field as the user types. The transformation runs on a 300ms debounce.

Solution

```
  Scenario: Slug auto-populates from title
    Given I am on "/articles/new"
    When I fill in "title" with "Hello World"
    Then "input[name=slug]" should have value "hello-world" within 1
second
```

For form inputs, `should have value` reads the live `.value` property (not the `value=` attribute), so it sees user input and JavaScript writes.

Recipe 352 Class Assertion With Budget

Problem

After clicking “Save”, a row gets the class `.row--saved` for a brief flash before the toast appears. Catching the transient class proves the visual feedback ran.

Solution

```
  Scenario: Row flashes saved class
    Given I am on "/inventory"
    When I click "Save" in the "SKU-12" row
    Then "tr[data-sku='SKU-12']" should have class "row--saved" within 1
second
```

The poll checks `element.classList.contains('row--saved')` every 100ms.

Recipe 353 Attribute Assertion

Problem

An `<img>` lazy-loads. Its `src` attribute starts as a 1x1 placeholder and becomes the real image URL once the image enters the viewport.

Solution

```
  Scenario: Lazy image swaps src
    Given I am on "/gallery"
    When I scroll to "img[data-id='hero']"
    Then "img[data-id='hero']" should have attribute "src" with value
"https://cdn.example.com/hero.jpg" within 5 seconds
```

For exact attribute matching. The check uses `element.getAttribute('src')`, so it sees both initial and JavaScript-updated values.

Recipe 354 Element Count Assertion

Problem

You filter a list down to all rows starting with “A”. You want to assert the list has exactly 7 results.

Solution

```
Scenario: Filter narrows to seven rows
  Given I am on "/customers"
  When I fill in "filter" with "A"
  Then ".customer-row" should have a count of 7 within 3 seconds
```

The matcher polls `await page.locator('.customer-row').count()` until it equals 7 or the budget elapses.

Tip Round-up

NEED	STEP
Element on screen	`"sel" should be visible`
Element gone	`"sel" should be hidden`
Element in DOM, any visibility	`"sel" should be attached`
Keyboard focus	`"sel" should be focused`
Click target	`"sel" should be enabled` / `disabled`
Type-able input	`"sel" should be editable`
Substring text	`"sel" should contain text "X"`
Exact text	`"sel" should have text "X"`
Input value	`"sel" should have value "X"`
CSS class	`"sel" should have class "X"`
Attribute value	`"sel" should have attribute "X" with value "Y"`
Count	`"sel" should have a count of N`
Viewport membership	`"sel" should be in the viewport`
Role visibility	`the "OK" button should be visible`

Every matcher accepts a trailing `within N seconds` clause. Default is 5 seconds. Use a larger budget for slow back-ends; never zero.

Web-first assertions retire two old habits: `wait for AJAX` before checks, and `I wait 3 seconds` filler. Both still exist for legacy code, but new tests should default to the matchers in this chapter.

Chapter 37: Iframe Testing

Iframes are the cockroaches of web testing. Hard to kill, hard to ignore, and hidden in places you least expect: payment widgets, embedded video players, ad units, third-party comment systems, OAuth consent pages. The DOM inside an iframe is a separate document. Standard selectors do not cross that boundary. Click an iframe's button with a normal step, and Playwright reports "element not found", because as far as the parent document is concerned, the button doesn't exist.

Webship-js ~2.0 ships first-class iframe support. You switch into a frame, interact normally, then switch back to the root document. All steps inside an iframe behave exactly as they do on a top-level page. The frame context is per-scenario, the next scenario starts fresh on the root document.

How Frame Switching Works

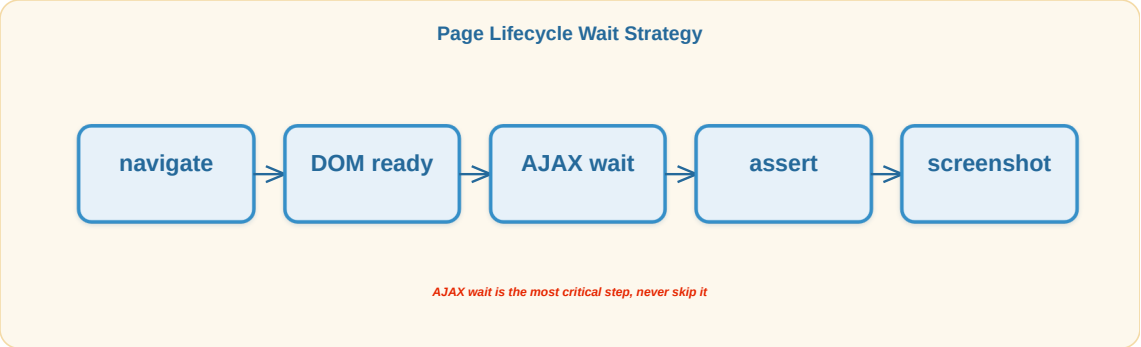


FIGURE 37.1 *How Frame Switching Works*

Every interaction step in webship-js targets a “current frame.” By default, that is the top-level document. When you switch to an iframe, the current frame becomes that iframe’s content. All subsequent clicks, fills, and assertions resolve against the iframe’s DOM. When you switch back to the root document, you return to the top-level page.

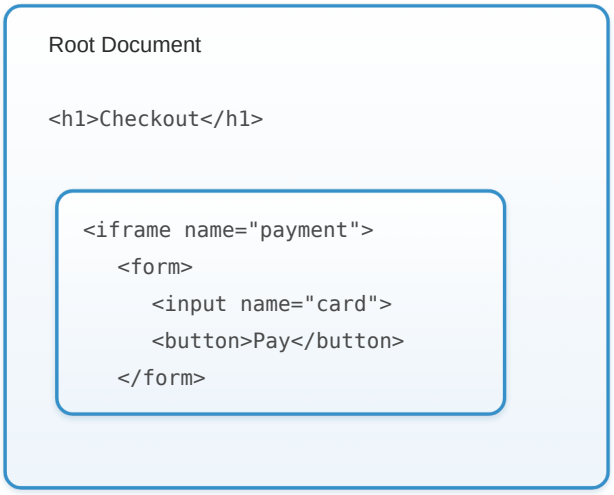


FIGURE 37.2 *How Frame Switching Works*

Without switching, `When I fill in "card" with "4242..."` fails. After `switch to the iframe with name "payment"`, the same step succeeds.

Under the hood, the switch steps are built on Playwright's `frameLocator`. When you switch to an iframe, webship-js stores a frame-scoped locator for the scenario, effectively `page.frameLocator('iframe[name="payment"]')`, and every later step resolves its selectors through that scope instead of the page. Two properties of `frameLocator` matter in practice. First, it is lazy: the iframe element is re-resolved on every action, so a widget that reloads itself mid-scenario does not leave you holding a stale frame reference. Second, it is strict about existence: if the iframe never attaches, the step fails with a clear "frame not found" instead of silently clicking the wrong document. Switching to the root document simply drops the stored scope, and the per-scenario reset means the next scenario always starts on the top-level page, no leakage between tests.

Recipe 355 Switching to an Iframe by Name

Problem

The Tuesday before release, your checkout suite goes red on every project that uses the new payment provider. The error says `cardNumber` cannot be found, and a teammate burns an hour "fixing" selectors that were never broken. The real cause: the provider ships its card input as `<iframe name="card-frame">`, and your fill steps are searching the parent document for a field that lives in a different one.

Solution

```
Feature: Checkout via payment iframe

  Scenario: Pay with card
    Given I am on "/checkout"
    When I switch to the iframe with name "card-frame"
    And I fill in "cardNumber" with "4242424242424242"
    And I fill in "expiry" with "12/30"
    And I fill in "cvc" with "123"
    And I switch to the root document
    And I click "Place Order"
    Then ".order-confirmation" should be visible within 10 seconds
```

The switch step builds `page.frameLocator('iframe[name="card-frame"]')` and stores it as the scenario's current frame. From that point, `I fill in "cardNumber"` runs the same label-first field resolution it always does, label, then id, name, placeholder, just scoped to the iframe's DOM instead of the page's. Auto-waiting still applies inside the frame: the fill waits for the field to be attached, visible, and editable before typing, and the AfterStep `smartSettle` still watches the parent page's AJAX counter and mutation timestamp, so a parent-side reaction to the iframe (a price recalculation, say) settles before the next step.

The `switch to the root document` step is critical. If you forget it, subsequent steps still target the iframe and will not find the "Place Order" button.

At scale: payment iframes are the most duplicated test code in a multi-project portfolio, every commerce site you run has the same switch-fill-fill-fill-switch dance. Put the iframe's selector in a shared JSON file under `tests/selectors/` (loaded through the `worldParameters.selectors` registry) and, if the provider is the same everywhere, wrap the whole dance in one custom step. When the provider renames its frame, you fix one file and twenty checkouts go green again.

Recipe 356 Switching to an Iframe by Title

Problem

The third project you onboard uses the same payment vendor as the first two, but a newer embed version, and this one ships its iframe with no `name` attribute at all. The only thing it carries is `title="Payment form"`, added for screen readers. Your by-name step from the last recipe has nothing to grab.

Solution

```
Scenario: Pay via iframe with no name attribute
  Given I am on "/checkout"
  When I switch to the iframe with title "Payment form"
  And I fill in "cardNumber" with "4242424242424242"
  And I switch to the root document
```

The step resolves to `page.frameLocator('iframe[title="Payment form"]')`, same scoping mechanics as the by-name variant, different attribute. `title` is the second most reliable iframe identifier after `name`, and for the same reason your accessibility audits care about it: an iframe's `title` is what a screen reader announces, so changing it is an a11y regression, not a refactor. Vendors and designers leave it alone. If your suite already runs `the page should pass an accessibility audit at level "AA"`, you get a free early warning, the audit flags a missing iframe title before your switch step ever fails on it.

Recipe 357 Switching by Locator

Problem

Then there is the iframe with neither `name` nor `title`, usually one your own front-end team generated, because nobody writes those attributes unless a linter makes them. The new hire's first PR targets it with `.css-x7k2j > iframe:nth-child(2)`, which works until the next style refactor. The only identifier that will survive a redesign is a CSS selector on something deliberate, like `iframe[data-component="stripe-card"]`.

Solution

```
  Scenario: Switch by CSS selector
    Given I am on "/checkout"
    When I switch to iframe with locator "iframe[data-component='stripe-card']"
    And I fill in "cardNumber" with "4242424242424242"
    And I switch to the root document
```

The locator string goes straight into `page.frameLocator(...)`, so anything Playwright accepts as a selector works here, CSS, attribute selectors, the lot. That makes this the escape hatch when `name` and `title` are missing, and the foot-gun when you feed it a generated class name. Prefer `data-*` attributes: they exist only for hooks like this, so no designer has a reason to rename them.

At scale: raw CSS strings inside feature files are where multi-project maintenance goes to die. Register the locator once in a shared selectors JSON file (`{"css": {"payment_frame": "iframe[data-component='stripe-card']"}}` in `tests/selectors/`, auto-loaded via `worldParameters.selectors.files`) and reference the name. When the vendor changes its embed markup, the diff that fixes all your projects is one line in one file, and the review takes thirty seconds instead of a selector hunt across twenty repositories.

Recipe 358 Clicking a Button Inside an Iframe

Problem

An order flow ends in a confirmation iframe: click “Confirm” inside it, then check the “Order accepted” message that renders in the same frame. Doing this with bare steps works, but six months later the scenario reads like every other click-and-check, and the reviewer approving a change cannot tell which steps run inside the frame and which on the page. You want the click, the assertion, and the frame boundary to be visible in the Gherkin itself.

Solution

```
Scenario: Confirm inside iframe
  Given I am on "/orders/new"
  When I switch to the iframe "confirmation"
  And I click "Confirm" inside the iframe
  Then I should see "Order accepted" inside the iframe
  And I switch to the root document
```

The `inside the iframe` suffix tells the step definition to resolve through the scenario’s current `frameLocator` rather than the page, and it tells the human the same thing. Mechanically, the click is a normal Playwright click with full actionability checks: the button must be attached, visible, stable, and enabled inside the frame before the click fires, with the usual web-first polling while it gets there. The suffix costs nothing at runtime; what it buys you is a feature file where the frame boundary is explicit, which is exactly the line a product owner or a reviewer needs when they ask “where does the third-party part start?”

Recipe 359 Asserting Text Inside an Iframe

Problem

Marketing swaps the live-chat vendor on a Friday. The new widget loads, the page looks fine, and nobody notices for two weeks that the chat bubble says nothing because the embed token was wrong, until a customer mentions it. A one-line smoke check that the widget's iframe actually renders "We're online" would have caught it on the first nightly run.

Solution

```
Scenario: Live chat shows online status
  Given I am on "/"
  When I switch to the iframe with title "Support chat"
  Then I should see "We're online" inside the iframe
  And I switch to the root document
```

The `inside the iframe` variant of `I should see` searches the iframe's body text through the stored `frameLocator`, not the parent page. Third-party widgets boot slowly, they load their own scripts, open their own connections, so the web-first polling matters here: the assertion retries every 100ms (up to the default 5 seconds) while the widget initializes, instead of failing on the first empty paint. No sleep, no flake.

At scale: the same chat widget, cookie banner, or survey embed usually appears on most of the sites you maintain. Keep these checks in one widget-smoke feature, tag the scenarios with something like `@third-party`, and reuse the file across projects with `LAUNCH_URL` pointing at each site. The tag earns its keep in CI: exclude `@third-party` from PR pipelines (a vendor outage should not block your developers' merges) and run it nightly, where a failure means an email to the vendor instead of a reverted release.

Recipe 360 Filling a Field Inside an Iframe by Attribute

Problem

An embedded comment widget renders a textarea with no label, no placeholder, no id, just `data-role="comment-body"`, because the vendor's accessibility story is "we'll get to it". Label-first resolution has nothing to latch onto, and you don't control the widget's markup, so fixing the HTML is not on the table. You need to fill and submit through the attribute you actually have.

Solution

```
  Scenario: Submit a comment via embedded widget
    Given I am on "/articles/42"
    When I switch to the iframe "comments"
    And I fill in "comment-body" with "Loved the article" inside the
iframe
    And I click "Post" inside the iframe
    Then I should see "Comment posted" inside the iframe
    And I switch to the root document
```

The combined `inside the iframe` step lets you target attribute-based fields without breaking out of the frame. Resolution still walks the usual chain, label, id, name, placeholder, then attribute values, it just walks it inside the iframe's document, so `comment-body` matches the `data-role` value even though no label exists. Everything else stays scenario-normal: the fill waits for editability, the click on "Post" waits for actionability, and the closing assertion polls for the success message inside the same frame. One scenario, one behaviour, frame boundary explicit at both ends.

Recipe 361 Clicking by Attribute Inside an Iframe

Problem

A sponsored iframe drops over your page's hero section, and until it is dismissed it intercepts clicks meant for the content underneath, so half your homepage scenarios fail with "element is covered" on days when the ad server is feeling generous. The close button inside the ad has no text, only an icon and `aria-label="Close ad"`. You need to dismiss it by that attribute, then prove the overlay is gone.

Solution

```
Scenario: Dismiss ad iframe
  Given I am on "/"
  When I switch to the iframe with title "Sponsored"
  And I click "Close ad" by attr inside the iframe
  Then "iframe[title='Sponsored']" should be hidden within 3 seconds
```

The `by attr` variant matches the click target against attribute values, `aria-label`, `title`, `data-*`, instead of visible text, which is exactly what icon-only buttons need. The closing assertion uses `should be hidden`, the web-first matcher that passes when the element is invisible *or* detached entirely; an iframe that removes itself from the DOM satisfies it either way, polled every 100ms until it does. One quirk to know: after clicking inside an iframe that removes itself, you would normally still need `switch to the root document`, but if the current frame is gone, the next step falls back to the root automatically rather than failing on a dead frame reference.

At scale: before you copy this recipe into twenty suites, ask whether the ad needs to exist in your tests at all. If the ad's behaviour is not what you are testing, the cheaper fix is one shared `Background` line that blocks the ad network's URL with the network-mock step from Chapter 39, no iframe ever loads, no overlay ever intercepts a click, and every page loads faster in CI. Reserve the click-to-dismiss pattern for the one suite where the ad experience itself is under test.

Recipe 362 Negative Iframe Text Assertion

Problem

A returning customer navigates back to the checkout page and the payment iframe greets them with the previous card number, pre-filled from a caching layer that should never have touched it. That is not a UI bug; that is the kind of incident that gets its own postmortem and a very uncomfortable compliance call. You want a standing smoke test that proves card digits never echo back into the payment frame after navigation.

Solution

```
Scenario: Card field empty after navigation
  Given I am on "/checkout"
  When I switch to the iframe "payment"
  Then I should not see "4242" inside the iframe
  And I switch to the root document
```

`should not see ... inside the iframe` is the negation matcher, scoped to the current frame's body text. It returns immediately when the text is absent and fails the moment it is found, which makes it nearly free to run, so there is no runtime excuse to skip it. One honest caveat: a negative text assertion proves absence *at the moment it runs*. Place it after the navigation that could trigger the leak, not before, or it passes without testing anything.

At scale: checks like this are evidence, not just tests. Tag them `@security` or `@compliance`, run the tag across every project nightly, and stamp the run with report metadata (`--metadata` flags on the report generator) so the cross-project dashboard carries a per-product "no sensitive data echoed" line. When an auditor, or your PMO, asks how you know card data never leaks into the page, "a tagged scenario asserts it on every site, every night" is an answer; "we reviewed the code in March" is not.

Recipe 363 Nested Iframes

Problem

A pricing widget arrives as an iframe, and somewhere inside it the vendor embedded *another* iframe for the live-chat trigger. Iframes in iframes, the test that needs to click “Talk to sales” has to cross two document boundaries, and the first time someone tries it with a single switch, the step fails and the debugging session runs long because the button is visible right there on the screen.

Solution

Webship-js currently supports one level of iframe switching at a time. To reach a nested iframe, switch to the outer first, then re-target the inner, the resolver matches against frames at every depth.

```
Scenario: Click chat trigger inside nested iframe
  Given I am on "/pricing"
  When I switch to the iframe "pricing-widget"
  And I switch to the iframe "chat-trigger"
  And I click "Talk to sales" inside the iframe
  And I switch to the root document
```

Each switch re-scopes from wherever you currently stand. The first switch stores a `frameLocator` for the pricing widget; the second resolves “chat-trigger” from inside that scope, chaining frame locators the way Playwright chains `frameLocator().frameLocator()`. Both hops stay lazy, each level re-resolves on every action, so a chat widget that re-renders its inner frame does not stale the chain. The single `switch to the root document` at the end unwinds the whole stack; you do not exit one level at a time.

> **NOTE**
>
> This pattern works because the iframe resolver searches all frames at every depth. For deeply nested chains, design your fixtures so the inner iframe has a unique name, a one-hop switch to a uniquely named frame is far less brittle than a multi-hop chain that breaks whenever the vendor restructures its embed.

Recipe 364 Returning to the Root Document

Problem

A scenario fails on `I click "Place Order"` with “element not found”, and the button is plainly there when you open the page yourself. Someone re-runs it, someone blames the environment, someone adds a longer timeout that changes nothing. The actual bug is three lines up: the scenario switched into the payment iframe and never switched back, so the click is hunting for “Place Order” inside a card widget that has no such button.

Solution

```
Scenario: Iframe round trip
  Given I am on "/checkout"
  When I switch to the iframe "payment"
  And I fill in "cardNumber" with "4242424242424242"
  And I switch to the root document
  And I click "Place Order"
  Then ".confirmation" should be visible within 10 seconds
```

`switch to the root document` drops the stored `frameLocator` and points the scenario back at the page object itself, the cheapest step in the suite, and the one whose absence costs the most afternoons. Forgetting it is the single most common iframe-test bug, and the symptom is always the same: “element not found” on a step that targets the parent page, right after an iframe block.

The frame scope is per-scenario, so the damage never leaks into the next scenario, which is exactly why the bug is confusing: scenario B passes while scenario A fails on a “missing” element that both of them use.

At scale: make the pairing a review convention, every `switch to the iframe` must have a matching `switch to the root document` in the same scenario, the way every opening brace has a closing one. This is mechanical enough to automate: a few lines of script over your `.feature` files can count switches per scenario and fail the pipeline on an unbalanced pair. Cheap lint, and it deletes an entire class of “works on my machine” tickets across every project that touches an iframe.

Iframe Cheat Sheet

STEP	USE CASE
`I switch to the iframe "name"`	iframe has a `name` attribute
`I switch to the iframe with title "X"`	iframe has only a `title`
`I switch to iframe with locator "sel"`	iframe identified by CSS
`I switch to the root document`	exit iframe context
`I click "X" inside the iframe`	click visible label
`I click "X" by attr inside the iframe`	click by attribute
`I fill in "X" with "Y" inside the iframe`	fill field
`I should see "X" inside the iframe`	text present
`I should not see "X" inside the iframe`	text absent

End every iframe block with `switch to the root document`. Treat it like closing a brace, non-negotiable.

Chapter 38: Clock Mocking and Time Travel

Time is the most reliable bug-injector in software. A test that passes Monday at 9am fails Friday at 11pm because the test ran past midnight in another timezone. A reminder fires “two minutes” early because the developer typed 120 instead of 120000. A scheduled job hits the new-year boundary and chooses 1970.

Webship-js ~2.0 adds a full **clock mocking** family. You can freeze the system time at a chosen instant, advance it by milliseconds, seconds, or minutes, and pause or resume the in-page clock. The mock affects `Date.now()`, `new Date()`, `setTimeout`, `setInterval`, and the page’s scheduling primitives.

This is the toolkit for testing date-sensitive flows: trial expiry, scheduled emails, “X minutes ago” labels, countdown timers, midnight rollover, leap-year edge cases.

How Clock Mocking Works

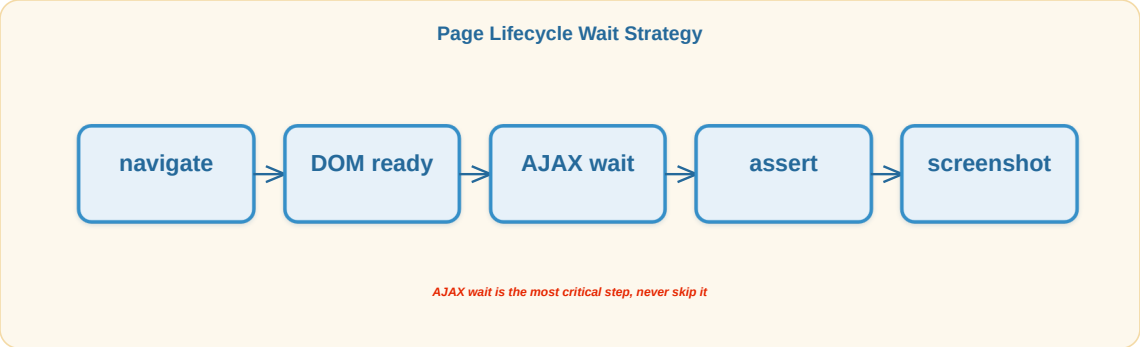


FIGURE 38.1 *How Clock Mocking Works*

Playwright (and therefore webship-js) installs a fake clock in the page context. The fake clock controls every time-reading API the page touches. When you advance the clock, callbacks scheduled with `setTimeout` and `setInterval` fire immediately instead of waiting in real time, you get a 30-day trial expiry to fire in a 100ms test run.

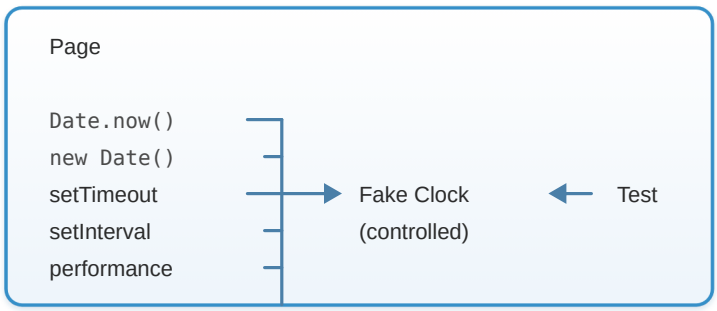


FIGURE 38.2 *How Clock Mocking Works*

Under the hood this is Playwright's `page.clock` API. Installing the fake clock injects a fake-timers implementation into the page *before any page script runs*, replacing `Date`, `setTimeout`, `setInterval`, `requestAnimationFrame`, and `performance.now` with controlled versions. That ordering is why the clock step comes before the navigation step in every recipe in this chapter: install after load, and the page's scripts have already read the real time. Webship-js wraps the whole API as a family of seven steps, install, advance, pause, resume, and friends, so the control surface lives in plain Gherkin.

Two control verbs do most of the work:

- **set the system time to "X"**, jump to an absolute instant
- **advance the clock by N ms/seconds/minutes**, step forward, firing scheduled callbacks along the way

One interaction worth knowing: webship-js's smart-wait also watches timers, via the `window.__webshipPendingTimers` wrapper. A paused clock can leave callbacks queued forever, but `smartSettle` never hangs on them, its budget is capped (1500ms after a step, 3000ms per navigation), so the worst a frozen timer costs you is that budget, not a stuck suite.

Recipe 365 Freezing System Time at a Known Instant

Problem

The sale-countdown test passes all day, then the 11pm nightly run fails because "23 hours" became "22 hours" while the suite queued. Worse, the same test asserts different strings depending on which timezone the CI runner woke up in. Any assertion that contains a number derived from "now" is a coin flip until you pin down what "now" is.

Solution

```
Feature: Sale countdown

  Scenario: 24-hour countdown displayed when 24h remain
    Given the system time is "2026-12-25T00:00:00Z"
    And I am on "/sale"
    Then ".countdown" should contain text "23 hours" within 2 seconds
```

`the system time is "X"` accepts any ISO-8601 timestamp and installs the fake clock at that instant, which is why it comes *before* `I am on "/sale"`. Install happens via an init script, so by the time the countdown code calls `Date.now()`, it already sees December 25th, midnight UTC. The clock is frozen there until you advance it explicitly. Notice the assertion says “23 hours”, not “24”: with exactly 24 hours remaining, a floor-based formatter displays 23, the off-by-one you only catch when you can pin the clock to the boundary. Network requests still happen in real time; only the in-page clock is mocked.

At scale: pick one canonical frozen instant for the whole organization, mid-quarter, midday UTC, nowhere near a DST switch or a month boundary, and use it as the default in every suite. When twenty projects each invent their own “today”, a date-formatting bug reproduces on three of them and the triage call wastes an hour discovering the difference was the fixture, not the code. One shared timestamp makes failures comparable across the portfolio.

Recipe 366 Advancing the Clock by Milliseconds

Problem

A toast notification auto-dismisses after exactly 3000ms. The first version of this test waited three real seconds and passed, then someone changed the dismiss delay to 5000ms, and the test kept passing because it asserted “hidden” *eventually*, on a generous timeout. You want a test that fails the moment the product’s 3000ms promise changes, and that does not spend three seconds of wall time per run proving it.

Solution

```
Scenario: Toast self-dismisses after 3 seconds
  Given the system time is "2026-01-01T00:00:00Z"
  And I am on "/"
  When I click "Save changes"
  Then ".toast" should be visible
  When I advance the clock by 3000 ms
  Then ".toast" should be hidden within 1 second
```

The toast's `setTimeout(dismiss, 3000)` sits queued in the fake clock. `I advance the clock by 3000 ms` runs the clock forward through that due time, firing the callback synchronously, three seconds of product time in roughly a millisecond of wall time. The step ordering carries the logic: assert the toast is visible *before* advancing (proving it showed), advance exactly 3000ms, then assert hidden. If a developer quietly changes the delay to 5000ms, the advance no longer reaches the callback and the test fails, which is precisely what you want from a test of a timing contract. Multiply the saved wall time across a thousand timing tests and you are buying back hours of CI per night.

Recipe 367 Advancing by Seconds

Problem

Every login flow with an OTP has one: the "Resend code" button that stays disabled for 60 seconds. Manual QA tests it by sending a code and literally watching the clock, once per release, per project, sixty seconds of a human's attention each time. And the automated version that waits a real minute is the slowest test in the suite by a factor of ten. The timer logic deserves a test that takes milliseconds.

Solution

```
Scenario: Resend button re-enables after 60 seconds
  Given the system time is "2026-01-01T00:00:00Z"
  And I am on "/verify"
  When I click "Send code"
  Then "#resend" should be disabled
  When I advance the clock by 60 seconds
  Then "#resend" should be enabled within 1 second
```

The countdown's `setTimeout(..., 60000)` fires instantly when the clock jumps forward 60 seconds, the fake clock walks through the due time and runs the callback before the step returns. The two web-first assertions around the jump do the real verification: `should be disabled` confirms the lockout engaged, `should be enabled within 1 second` confirms the re-enable handler ran. That trailing `within 1 second` is not for the timer (the timer already fired); it covers the page's own re-render after the callback.

At scale: OTP, resend, and rate-limit lockouts appear in every product that has a login, and they are usually copy-pasted timer code, which means they break in the same way everywhere. Keep one shared feature for the lockout behaviour, drive the base URL per project with the `LAUNCH_URL` env var, and the same five-line scenario verifies the contract on every site you run. When security tightens the lockout from 60 to 90 seconds, the change is one number in one feature file, not an archaeology dig across twenty repos.

Recipe 368 Advancing by Minutes

Problem

"The editor ate my draft" is the bug report nobody can reproduce, because reproducing it means typing for six minutes and watching whether auto-save actually fires on schedule. The auto-save interval is 2 minutes; the question is not whether it fires once but whether it keeps firing, interval

bugs love to fire the first tick and silently stop. You want consecutive save cycles verified in a test that finishes in under a second.

Solution

```
Scenario: Draft auto-saves every 2 minutes
  Given the system time is "2026-01-01T00:00:00Z"
  And I am on "/articles/new"
  When I fill in "title" with "Hello"
  And I advance the clock by 2 minutes
  Then ".save-status" should have text "Saved" within 1 second
  When I fill in "title" with "Hello World"
  And I advance the clock by 2 minutes
  Then ".save-status" should have text "Saved" within 1 second
```

Each `advance the clock by 2 minutes` runs the fake clock through exactly one interval tick, firing the auto-save callback once, not zero times, not twice. That is the property real-time tests cannot pin down: with a real clock, you never know whether your 2-minute wait caught one tick or two. Here the arithmetic is exact. The edit-advance-assert rhythm also catches the “fires once then stops” class of interval bug, because the second cycle is asserted independently of the first. If the second “Saved” never appears, you have found a cleared interval or a debounce that swallowed the tick.

Recipe 369 Pausing the Clock

Problem

A relative-time label reads “5 seconds ago” when the page renders, and “6 seconds ago” by the time your assertion runs, because real time kept moving between render and check. The test flakes on slow CI runners and passes locally, which is the most expensive kind of flake to diagnose. You want time to stand completely still while you assert, and to move only when the test says so.

Solution

```
  Scenario: Relative-time label is clock-driven
    Given the system time is "2026-01-01T00:00:05Z"
    And the active comment timestamp is "2026-01-01T00:00:00Z"
    And I pause the clock
    When I am on "/comments"
    Then ".comment .relative-time" should have text "5 seconds ago"
    When I advance the clock by 60 seconds
    Then ".comment .relative-time" should contain text "1 minute" within 1
second
```

With the clock paused, `Date.now()` returns the same value no matter how slowly the CI runner grinds through the steps, the “5 seconds ago” assertion cannot drift into “6 seconds ago”, because in-page time literally does not pass between steps. The label then changes *only* in response to `advance the clock`, which is the behaviour under test: the page recomputes relative times from the clock, and the test owns the clock. `pause the clock` is implicit after `set the system time`, listing it explicitly costs one line and makes the intent obvious to whoever reads the scenario next. One more thing to notice: `the active comment timestamp is ...` is not a web-ship-js step. It is an app-specific fixture step you write yourself (Chapter 17 territory) to seed the data the page renders, the clock steps control time, but your fixtures still control state.

Recipe 370 Resuming the Clock

Problem

You froze time to get the dashboard into a known state, “yesterday’s data, 24 hours stale”, but the second half of the test clicks “Refresh data” and waits on a real network round trip. A frozen clock and a real network make awkward partners: the request happens in wall time while the page’s own timers stay parked. You want deterministic setup, then a clean handover back to real time for the live part.

Solution

```
Scenario: Resume real time for network assertions
  Given the system time is "2026-01-01T00:00:00Z"
  And I am on "/dashboard"
  When I advance the clock by 24 hours
  And I resume the clock
  And I click "Refresh data"
  Then ".data-loaded" should be visible within 5 seconds
```

Resuming releases the page back to wall time: timers tick on their own again and `Date.now()` moves without your help. One subtlety worth holding onto, resume does not *rewind*. The page still believes it is 24 hours later than when the scenario started; what changes is that time flows forward from there at the normal rate. So use resume when the remaining steps depend on things actually happening in real time (network latency, animations you want to watch in a recorded video), and stay on the fake clock for everything else. If you find yourself resuming just to "make a wait work", that is usually a smell, the web-first `within N seconds` assertions poll in wall time regardless, so they work fine under a mocked clock.

Recipe 371 Settling Pending Timers

Problem

The animations team staggers three callbacks, 100ms, 500ms, 1000ms, and next sprint they retune the timings, because that is what animation teams do. The test that advanced the clock by exactly 1000ms breaks on every retune, and the review thread fills with the eternal question: "how long should we advance?" You want a step that means "fire everything that is queued, whatever the delays happen to be this week."

Solution

```
Scenario: All pending timers fire
  Given the system time is "2026-01-01T00:00:00Z"
  And I am on "/"
  When I click "Start all animations"
  And I wait until pending timers settle
  Then ".animation-done" should have a count of 3 within 1 second
```

`wait until pending timers settle` advances the fake clock just far enough to run every queued callback, in scheduled order, regardless of what the delays are, fast-forward-everything semantics. It is the clock-mock sibling of the smart-wait machinery: where `smartSettle` *observes* `window.__webshipPendingTimers` and waits for the count to reach zero in real time, this step actively drains the queue by moving time itself. The decision rule between this recipe and the previous three: advance by an exact amount when the delay *is* the contract you are testing (a 60-second lockout must be 60 seconds); settle everything when the delays are incidental (animation polish that retunes every sprint).

At scale: adopt that decision rule as a written convention and the "how long should we advance?" review argument disappears across all your projects. Hardcoded advance values in feature files should always trace to a documented product number, if a reviewer cannot find where the spec says 60 seconds, the scenario should be settling timers instead. That single convention keeps twenty suites from breaking every time a front-end team retunes its easing curves.

Recipe 372 Testing a Daily Cron Boundary

Problem

The daily quota resets at midnight UTC, which means your users in Auckland hit the rollover during their workday and find the bugs first, while your developers, asleep nine timezones away, never see the boundary at all. "Wait until tomorrow and check" is not a test plan. You need to stand at 23:59:50 and step across midnight on demand, as many times per day as the suite runs.

Solution

```
  Scenario: Quota resets at UTC midnight
    Given the system time is "2026-03-14T23:59:50Z"
    And I am on "/account"
    Then ".quota-remaining" should have text "3 / 5 actions"
    When I advance the clock by 15 seconds
    Then ".quota-remaining" should have text "5 / 5 actions" within 1
second
```

Freeze ten seconds before midnight, assert the old day's state, then cross 00:00:00 in a single 15-second advance. Any interval the page runs ticks during that advance, reads the new date from the mocked `Date`, and refreshes the counter, the same code path a real user's open tab would execute at midnight, compressed into milliseconds. The first assertion is load-bearing: it proves the *pre*-rollover state, so when the second assertion passes you know the change happened because of the boundary, not because the page rendered fresh.

At scale: date boundaries are a category, not a one-off, midnight rollover, month-end, December 31st, February 29th, the DST switches. Keep a small "date edges" feature per product that pins each gnarly instant with a frozen clock, and run it nightly everywhere. It is a few dozen lines per project, and it converts "we think the leap-year path works" into a green row on the cross-project report, the difference between a guess and an answer when someone asks in February.

Recipe 373 Testing a Trial Expiry Banner

Problem

Trial expiry is checked on page load: "expires in 1 day" today, "Trial expired" tomorrow. The support tickets cluster exactly at that boundary, users who swear they had a day left, billing flows that locked a day early. QA cannot verify the fix without either waiting out a real seven-day trial or asking a developer to fiddle the database date by hand, once per project, every time the logic changes.

Solution

```
  Scenario: Trial banner flips after 7 days
    Given the system time is "2026-01-01T00:00:00Z"
    And the active trial started at "2025-12-25T00:00:00Z"
    And I am on "/dashboard"
    Then ".trial-banner" should contain text "Trial expires in 1 day"

  Scenario: Trial expires
    Given the system time is "2026-01-02T00:00:00Z"
    And the active trial started at "2025-12-25T00:00:00Z"
    And I am on "/dashboard"
    Then ".trial-banner" should contain text "Trial expired" within 2
seconds
```

Two scenarios, two different “today” values, one fixed trial-start fixture, same code path. Because the banner logic runs at load time, no advancing is needed, each scenario installs its chosen instant *before* navigation, so the page’s expiry calculation reads the mocked date on its very first `new Date()`. The pair brackets the boundary: one day before expiry and the day after. If you want to be thorough, add the exact-boundary scenario at `2026-01-01T00:00:00Z`, off-by-one bugs live precisely there, in the difference between `>` and `>=`. As in Recipe 369, `the active trial started at ...` is your own fixture step seeding app state; the clock steps only control what the page thinks “now” is.

Recipe 374 Avoiding Real Sleeps in Date Tests

You inherit a suite where someone solved every timer with a real wait. It worked when there were five tests. Now there are a hundred, the nightly run takes two hours, and most of that time is the suite doing literally nothing, sitting out countdowns that a mocked clock would cross instantly.

Anti-pattern

```
Scenario: BAD -- waits 60 real seconds
  Given I am on "/verify"
  When I click "Send code"
  And I wait for 60 seconds
  Then "#resend" should be enabled
```

This test runs for 60 seconds of wall time. In a suite of 100 such tests, that's 100 minutes of CI burn for nothing.

Better

```
Scenario: GOOD -- advances mock clock
  Given the system time is "2026-01-01T00:00:00Z"
  And I am on "/verify"
  When I click "Send code"
  And I advance the clock by 60 seconds
  Then "#resend" should be enabled within 1 second
```

The lesson is simple: any time you reach for `I wait for N seconds`, ask whether `I advance the clock by N seconds` would do the same job in zero wall time. This is also the house rule of webship-js itself, the framework ships with no static sleeps anywhere; every built-in wait goes through `smartSettle`, which watches AJAX counts, pending timers, and DOM mutations instead of the wall clock. A real 60-second wait is not even safer: it still races the thing it waits for, it just races slowly. The clock advance does not race anything, it *causes* the event, then asserts.

At scale: static sleeps are the cheapest suite-wide cost to eliminate, and the easiest to enforce mechanically. Grep your feature files for `I wait for` in the CI pipeline and fail the lint when one appears outside an approved allowlist. The arithmetic sells itself to whoever owns the CI bill: a hundred 60-second sleeps is 100 minutes of farm time per run, per project, every night, across 40 suites, that is a number with a currency sign on it, spent on waiting.

Common Clock Patterns

GOAL	PATTERN
Freeze for a deterministic page snapshot	`Given the system time is "X"`
Fire scheduled callback now	`When I advance the clock by N ms`
Cross a calendar boundary	`When I advance the clock by N hours`
Fast-forward everything queued	`When I wait until pending timers settle`
Stop the in-page clock from drifting	`And I pause the clock`
Hand back to real time	`When I resume the clock`

Use the clock mock anywhere your test logic depends on “what time is it now.” Real time is for end-to-end smoke tests and human users. Test time is yours to control.

Chapter 39: Network Mocking and Request Recording

A test suite that hits real APIs is at the mercy of real APIs. The payment gateway times out at 3am. The third-party search backend returns a 500 because someone deployed bad config. The stock-data provider returns yesterday's data because a cache hasn't refreshed. None of these are your bugs, but every one of them turns your green build red.

Webship-js ~2.0 ships a **network mocking** family that lets you control exactly what the browser sees over the wire. You can block requests, delay them, force them to return specific status codes, or return arbitrary JSON. Combined with the **network recording** family, you can also assert what requests the page actually fired, useful for tracking analytics, GTM events, or back-end RPC calls.

This chapter covers the two halves: shaping what the page receives, and observing what the page sends.

When to Mock and When Not To

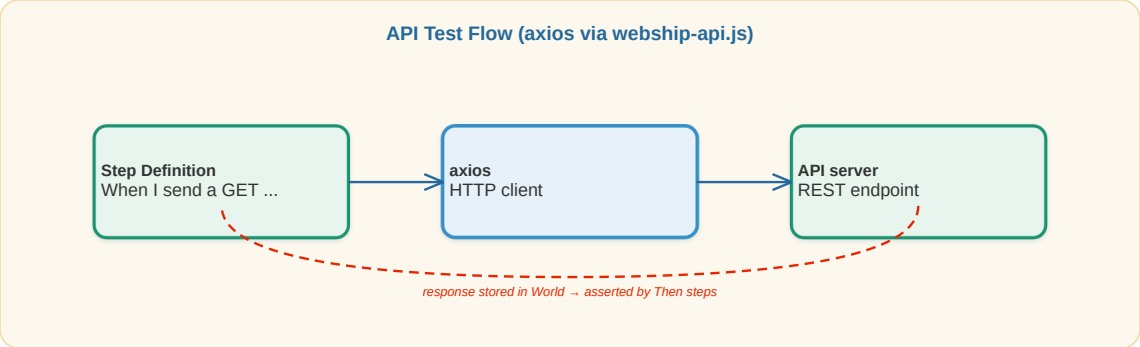


FIGURE 39.1 *When to Mock and When Not To*

Mock when:

- The dependency is flaky, slow, or rate-limited
- You need to test error paths (`500`, `503`, network timeout)
- The dependency returns non-deterministic data (timestamps, random IDs)
- You don't control the dependency's environment

Don't mock when:

- The test is an end-to-end smoke test of a real production flow
- You're testing the integration itself, not the page's behaviour
- Stubbing would hide a real bug in your own service

Most browser-side tests fall in the "mock" bucket. Webship-js makes mocking trivial enough that it's the default, not the exception.

The machinery underneath is Playwright's `page.route()`. Every mock step registers a route handler that intercepts matching requests inside the browser's own network stack, before anything leaves the machine, and either fulfills them with your canned response, aborts them, or holds them. URL matching uses glob patterns: `*` matches within a path segment, `**` matches across everything, including the scheme and host. That second one is the pattern you will type constantly.

`"**/api/stats"` matches `http://localhost:8080/api/stats` on your laptop and `https://staging.example.com/api/stats` in CI, because the `**` swallows the part that differs per environment. A bare `"/api/stats"` matches almost nothing, get in the habit of leading with `**/`.

Recipe 375 Returning a Mock Status Code

Problem

A user leaves the dashboard open overnight, comes back with coffee, clicks something, and the session has long expired. The page should bounce them to `/login`; instead, half the time it renders a broken dashboard skeleton over a wall of console errors. QA cannot reproduce it on demand, because reproducing it honestly means letting a real session age out. The expiry redirect ships untested release after release for exactly that reason.

Solution

```
Feature: Session-expiry redirect

  Scenario: 403 on profile triggers login redirect
    Given the URL "**/api/profile" returns status 403 with body ""
    When I am on "/dashboard"
    Then I should be on "/login"
```

The mock registers a `page.route()` handler for the glob `**/api/profile` and fulfills any matching request with a 403 and an empty body, the request never leaves the browser, no server is involved, no session needs to exist or expire. The leading `**/` is what makes the rule portable: it matches the path regardless of host, so the same scenario works against `localhost` and staging without edits. An empty body is fine here because the page only branches on the status code; when the UI also renders the error payload, give the mock a real body (next recipe). The closing `I should be on "/login"` then verifies the redirect landed.

At scale: error paths are the least hand-tested code you own, nobody manually expires a session on twenty products before a release. Put scenarios like this in a shared error-path feature, tag them `@error-paths`, and run the tag everywhere with the base URL coming from the `LAUNCH_URL` env var. The first full run is illuminating: you will learn which of your products redirect gracefully and which serve a white screen, and your delivery manager finally gets that answer as a table instead of a shrug.

Recipe 376 Returning a Mock Status With Body

Problem

When a card is declined, the payment API answers `402 Payment Required` with the decline reason in the body, and the checkout page is supposed to surface that reason to the shopper. Testing it against the payment sandbox means maintaining magic test-card numbers that trigger specific declines, and the sandbox is down or rate-limited the week you need it most. You want the decline path on demand, with the exact message you choose.

Solution

```
  Scenario: 402 with message displays card-declined banner
    Given the URL "**/api/payment" returns status 402 with body "Card
declined: insufficient funds"
    When I am on "/checkout"
    And I click "Place Order"
    Then ".error-banner" should contain text "insufficient funds" within 3
seconds
```

The route handler fulfills the request with both pieces the page needs: the 402 status that triggers the error branch, and the plain-text body the branch renders. This tests one level deeper than the previous recipe, not just “the page notices the failure” but “the page shows the user *why*”. That distinction catches a classic bug: error handlers that swallow the response body and show a generic

“Something went wrong”, which is precisely what you do not want a shopper to see when the fix is “use another card”. The body here is plain text; when the page parses the error response as JSON, use the docstring form in the next recipe instead.

Recipe 377 Returning a Mock JSON Body

Problem

The dashboard renders a card per stat from `/api/stats`, and every test against it asserts on numbers that change by the minute. The suite passes at 10am and fails at noon because `activeNow` moved. Then comes the demo where staging had no data and the revenue card proudly displayed zero in front of a client. Live numbers belong in production; tests need numbers that hold still.

Solution

```
Scenario: Dashboard renders mocked stats
  Given the URL "**/api/stats" returns the JSON:
    """
    {
      "totalUsers": 1024,
      "activeNow": 37,
      "revenueToday": 4250.50
    }
    """
  When I am on "/dashboard"
  Then ".stat-users" should have text "1,024" within 2 seconds
  And ".stat-active" should have text "37"
  And ".stat-revenue" should contain text "4,250.50"
```

The docstring body is served back with `Content-Type: application/json` and a 200 status, straight from the route handler, the page’s `fetch` parses it exactly as if the stats service had answered. From the front-end code’s point of view there is no way to tell the difference, which is the

point: you are testing the *rendering* (does 1024 become "1,024"? does 4250.5 become "4,250.50"?), not the stats service. Deterministic data is what makes exact-text assertions possible; against live data you would be stuck with weak "should be visible" checks that pass even when the formatting breaks.

At scale: the payload in the docstring is a contract, and on a multi-project team it pays to treat it like one. Agree on a canonical fixture shape per API and reuse it in every suite that mocks the endpoint, when the schema gains a field, one review updates the fixtures and you see immediately which projects' UIs care. Deterministic mocks also make products comparable: the same mocked stats rendered by five different dashboards is the cheapest cross-project regression harness you will ever build.

Recipe 378 Blocking a URL Entirely

Problem

Someone profiles a slow test run and finds the suite spends 800ms of every page load waiting on `analytics.js` from a third-party CDN, a script whose entire contribution to your tests is making them slower and occasionally flakier when the CDN hiccups. Multiply 800ms by every navigation in every scenario and the analytics vendor is quietly your largest CI line item. You want it gone from test runs without touching the page code.

Solution

```
Scenario: Analytics-free pageload
  Given the URL "https://cdn.example.com/analytics.js" is blocked
  When I am on "/"
  Then "h1" should have text "Welcome" within 2 seconds
```

`is blocked` registers a route that calls `route.abort()`, the browser sees an immediate network failure instead of a slow round trip. For an async script tag, that failure is a silent no-op; the page carries on without it. The pattern here is an exact URL, which blocks that one file; to silence

the vendor wholesale, widen the glob to `"https://cdn.example.com/**"` and every request to that host dies at birth. One warning before you get block-happy: abort a script the page *hard-depends* on and you have not sped up the test, you have changed what it tests, block the optional, mock the essential.

At scale: a shared `Background` that blocks the usual suspects, analytics, tag managers, chat bubbles, font CDNs, ad slots, is the single cheapest speedup available to a team running forty suites nightly. It compounds three ways: faster page loads (CI minutes are money), fewer third-party flakes (a CDN outage stops failing your releases), and faster settles (webship-js's smart-wait watches `window.__webshipAjaxCount`, and blocked requests fail instantly instead of dribbling in for seconds). Keep the block list in one shared feature or helper so all projects inherit updates together.

Recipe 379 Delaying a URL

Problem

The designer spent a sprint on the search loading state, skeleton, spinner, the works, and asks whether anyone has actually verified it. Nobody has, because nobody *can*: the staging backend answers in 50ms and the spinner is gone before a human or a screenshot catches it. Loading-state code is real code with real bugs (spinners that never hide are a classic), and it needs a network slow enough to expose it.

Solution

```
Scenario: Spinner is visible during slow request
  Given the URL "**/api/search**" is delayed by 2000 ms
  When I am on "/"
  And I fill in "q" with "playwright"
  And I click the "Search" button
  Then ".spinner" should be visible within 1 second
  And ".spinner" should be hidden within 3 seconds
```

`is delayed by 2000 ms` holds the matched request inside the route handler for two seconds, then lets it continue to the real backend, this is a delay, not a stub, so the response is still genuine. The trailing `**` on the glob matters here: search requests carry query strings (`/api/search?q=playwright`), and without the wildcard the pattern misses them and the request sails through at full speed. The two assertions then bracket the lifecycle: the spinner must appear well inside the 2-second delay window, and must be gone within a second of the response landing. A spinner that never hides, the bug users actually report, fails the second line. Note that the AfterStep smart-settle will not absorb the delay for you: its per-step budget is 1500ms, less than the 2000ms hold, so the explicit web-first assertions are doing the synchronization work here, exactly as they should.

Recipe 380 Asserting a Request Was Made

Problem

The nastiest front-end failures are the silent ones: a user clicks “Subscribe”, the success toast appears, and nothing was ever sent, a handler got unbound in a refactor, or a `preventDefault` ate the submit. The UI looks perfect; the mailing list stays empty; marketing notices three weeks later. Asserting on what the page *shows* cannot catch this class of bug. You have to assert on what the page *sends*.

Solution

```
Scenario: Subscribe button fires POST
  Given I start recording network requests
  When I am on "/"
  And I fill in "email" with "fan@example.com"
  And I click "Subscribe"
  Then a POST request to "**/api/subscribe" should have been made
```

`I start recording network requests` attaches a listener that logs the method and URL of every request the page fires, fetch, XHR, scripts, images, beacons, from that step until the scenario ends. The `Then` scans the log for a POST whose URL matches the glob `**/api/subscribe`.

The assertion is method-aware (`GET`, `POST`, `PUT`, `PATCH`, `DELETE`), which is half its value: a subscribe form that fires a GET with the email in the query string is a different bug, and one this assertion also catches. Start the recording *before* the action that should trigger the request, requests fired before the recording step simply do not exist as far as the log is concerned.

At scale: "did the beacon fire" is a question every product team asks about analytics, tag managers, and conversion pixels, usually in a panic, quarterly. A small `@tracking` feature per project, built on recorded-request assertions, replaces the panic with a nightly green row. It also gives product owners something they have rarely had: readable, self-verifying evidence that the measurement they base decisions on is actually wired up.

Recipe 381 Asserting Any-Method Request

Problem

Last quarter the health probe was a GET; then someone optimized it to a HEAD to save bandwidth, and the test that asserted `a GET request to "**/api/health"` went red on a change that broke nothing. The behaviour you actually care about is "the page checks its backend on load", the verb is an implementation detail that the front-end team is allowed to change without filing a ticket with QA.

Solution

```
Scenario: Health probe runs on load
  Given I start recording network requests
  When I am on "/"
  Then a request to "**/api/health" should have been made
```

Dropping the method from the assertion makes it verb-agnostic: the recorder logs every request's method and URL, and the method-less form matches on URL alone, GET, HEAD, POST, whatever the probe happens to be this quarter. This is the "test behaviour, not implementation" rule ap-

plied to the network layer: pin the verb only when the verb *is* the contract (a POST that creates something), stay loose when it is not (pings, beacons, telemetry). Tests that encode incidental detail are the ones that train teams to ignore red builds.

Recipe 382 Asserting No Request Was Made

Problem

Your privacy mode promises “no analytics”, and legal would like proof rather than promises, because one stray beacon after a user opts out is not a bug ticket, it is a regulatory incident with a fine attached. The dangerous regression here is invisible by design: the page looks identical whether the tracker fires or not. The only place the truth lives is on the wire, so that is where the assertion has to live.

Solution

```
  Scenario: Privacy mode silences analytics
    Given I start recording network requests
    When I am on "/"
    And I click "Enable privacy mode"
    And I click "Subscribe"
    Then no request to "https://analytics.example.com/**" should have been
made
```

The negative form scans the same recording and fails if *any* request matches the glob. Pattern width is everything in a negative assertion: trackers append query strings (`/track?event=subscribe&uid=...`), and a too-narrow pattern that misses them makes the test pass while the beacon fires, a vacuous green, the worst kind. The `/**` suffix closes that gap by matching anything on the host. Ordering matters for the same reason: perform every action that *might* fire the tracker before the `Then`, or you are asserting silence over an empty stretch of the scenario. For belt-and-braces, combine with `is blocked`: the block guarantees no data escapes even if the bug exists, and the recording assertion still fails the build because the *attempt* is logged.

At scale: consent and privacy assertions are evidence your PMO and legal team can actually use, but only if they run everywhere and someone can find the results. Tag them `@compliance`, run the tag across every project nightly, and stamp the report with metadata flags (`--env`, `--app-version`, `--metadata Consent=verified`) so the cross-project dashboard shows opt-out coverage per product, per version. When the regulator letter arrives, "here is last night's run on that exact version" beats any amount of meeting notes.

Recipe 383 Mocking Search Results

Problem

A search page makes a JSON request to `/api/search?q=...`. You want predictable results for an autocomplete test without worrying about real search-engine quirks.

Solution

```
  Scenario: Autocomplete shows top results
    Given the URL "/api/search" returns the JSON:
      """
      {
        "results": [
          {"id": 1, "title": "Playwright in Practice"},
          {"id": 2, "title": "Playing Cucumbers"},
          {"id": 3, "title": "Playgrounds for Developers"}
        ]
      }
      """
    When I am on "/"
    And I fill in "q" with "play"
    Then ".autocomplete .result" should have a count of 3 within 1 second
    And ".autocomplete .result" should contain text "Playwright in
Practice"
```

A `q=...` query string variant matches the exact path; webship-js matches against the URL substring by default, so any request whose URL includes `/api/search` is served the mock.

Recipe 384 Combining Block + Mock for Robust Tests

Problem

Your page calls a primary API and a fallback API. You want to force the fallback path by blocking the primary and mocking the fallback.

Solution

```
Scenario: Fallback path serves response when primary fails
  Given the URL "/api/v2/items" is blocked
  And the URL "/api/v1/items" returns the JSON:
    """
    {"items": [{"id": "legacy", "name": "Legacy Item"}]}
    """
  When I am on "/items"
  Then ".item" should have text "Legacy Item" within 2 seconds
```

Two mock rules, one scenario. The page tries `/api/v2/items`, gets blocked, falls back to `/api/v1/items`, and renders the mocked legacy data.

Recipe 385 Simulating Offline Mode

Problem

A page should show “You’re offline” when the browser detects no network. You want to test the banner without unplugging your laptop.

Solution

```
Scenario: Offline banner appears when network drops
  Given I am on "/"
  When the network is offline
  Then ".offline-banner" should be visible within 2 seconds
  When the network is online
  Then ".offline-banner" should be hidden within 2 seconds
```

`network is offline` flips the browser's `navigator.onLine` flag and fails all requests until you restore connectivity with `network is online`.

Recipe 386 Asserting Request Method Counts

Problem

A single page-load should fire exactly one POST to `/api/track`. Two POSTs would be a double-firing bug.

Solution

```
Scenario: Tracking fires exactly once per page-load
  Given I start recording network requests
  When I am on "/article/42"
  Then a POST request to "/api/track" should have been made
  When I am on "/article/42"
  Then a POST request to "/api/track" should have been made
```

Each navigation triggers a fresh check. The recording is per-scenario; switching scenarios resets the recorder.

> **TIP**
>
> Network mocking is per-scenario. If you mock `/api/x` in scenario A, scenario B starts with the real URL again. Use a `Background:` block to share mocks across scenarios in a feature.

When Network Mocking Is the Wrong Tool

Mocking is powerful but lossy, it hides real failure modes. If your test suite never sees the real API, you'll miss schema drifts, auth issues, and rate-limit bugs.

A healthy test pyramid runs:

- **Many unit tests** with full mocking
- **Many BDD browser tests** with selective mocking (mock unstable dependencies, real-test core flows)
- **A small end-to-end smoke suite** with zero mocking, hitting a staging environment

Webship-js can serve both the middle and the smoke layer, pick the right amount of mocking per suite.

Chapter 40: File Downloads and Video Recording

Two features that look unrelated, but both deal with the same thing: artifacts that leave the browser and live on the filesystem. Webship-js ~2.0 added first-class support for asserting that the browser downloaded a file (and what's inside it), plus the ability to record video of the test run for visual evidence.

Downloads are the more practical of the two, most teams have at least one CSV export, one PDF generator, or one "Save as ZIP" button that needs a test. Video recording is the more dramatic, when a flake hits in CI, a 30-second video of the test is worth a thousand log lines.

How Downloads Work in Webship-js

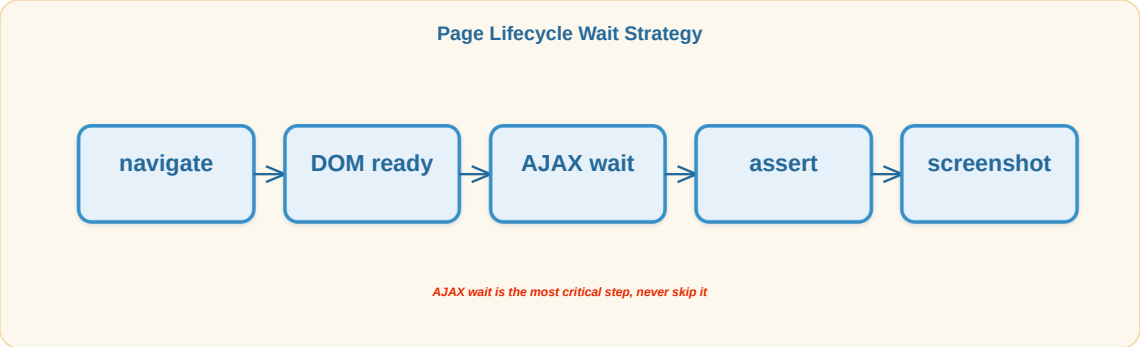


FIGURE 40.1 *How Downloads Work in Webship-js*

A “download” in browser terms is any response that triggers the browser’s save dialog, usually because of a `Content-Disposition: attachment` header or a programmatic anchor with `download="..."`. Playwright surfaces this as a `download` event on the page, and that’s what webship-js builds on: the download step arms a listener for that event *before* it triggers the link or URL, so there is no race between “click happened” and “download started”. The browser never shows a save dialog (there is no dialog in headless mode anyway); the bytes go straight to a temp directory that Playwright manages, and webship-js keeps a handle to the captured file on the scenario’s world object.

You don’t need to manage paths. The assertion steps refer to “the downloaded file”, the most recent download in the scenario. Because every scenario gets a fresh browser context, a download captured in one scenario can never bleed into the next. The suggested filename comes from the server’s `Content-Disposition: filename=` header (or the anchor’s `download` attribute), which means the name assertions are really *server contract* assertions, they catch the backend change, not just the frontend one.

Recipe 387 Downloading a File from a Link

Problem

The “Download CSV” link on your reports page has been there since the first release, and nobody has a test for it, it always just worked. Then a backend refactor renames the export controller, the link starts returning an HTML error page with a `.csv` name, and the first person to find out is the finance lead doing month-end close. Export links break silently because no human clicks them between releases. You want the suite to click this one every night.

Solution

```
Feature: Reports export

  Scenario: CSV export downloads with the right name
    Given I am on "/reports/sales"
    When I download the file from the link "Download CSV"
    Then the downloaded file name should be "sales-2026-05.csv"
```

The download step does two things in the right order: it arms Playwright’s `download` event listener first, *then* clicks the link by its visible text. That ordering matters, if you clicked first and listened second, a fast server could finish the download before the listener existed, and you’d wait forever. Once the event fires, the bytes land in a Playwright-managed temp directory and webship-js holds the file handle for the rest of the scenario. The name assertion checks the suggested filename, the one the server sent in `Content-Disposition`, with an exact match. Every later download-related `Then` step refers to this same file until the next download replaces it.

At scale: almost every project in a portfolio has at least one export, CSV reports, PDF invoices, ZIP backups, and they are the least-clicked links in manual QA. Tag these scenarios `@downloads` across all your suites so one nightly CI job can run every export check in the portfolio. The exact-name assertion doubles as a contract test on the backend: when a server-side rename slips through code review, this is the test that names it before finance does.

Recipe 388 Downloading by Partial File Name

Problem

The first version of this test asserted the exact name `invoice-2026-05-31.pdf`. It passed for exactly one day. The server stamps every download with the current date, so any exact-name assertion is a time bomb, it detonates at midnight, in the next month, or on the next billing cycle, and someone burns a morning "debugging" a test that was wrong from the start. You know the stable part of the name; assert only that.

Solution

```
Scenario: Invoice downloads with timestamped name
  Given I am on "/account/billing"
  When I download the file from the link "Download invoice"
  Then the downloaded file name should contain "invoice-"
```

`should contain` runs a substring match against the suggested filename instead of exact equality. The capture mechanics are identical to Recipe 387, listener armed before the click, file held on the scenario, only the comparison relaxes. Use it whenever the server bakes dates, user IDs, or random tokens into names. The right discipline is to assert the parts the *contract* guarantees: the prefix that identifies the document type and, in a second assertion if you like, the extension.

Pro Tip: write this rule into your team's test style guide: "never pin a generated token in a filename assertion, assert prefix and extension." It's the kind of trap every new hire falls into once. A one-line convention saves you the recurring midnight-rollover failure across every project that exports anything.

Recipe 389 Downloading Directly from a URL

Problem

Not every download hangs off an anchor you can click. Your dashboard builds the export URL in JavaScript and fires it from a button handler, there is no `<a>` element, and the link-based step has nothing to grab. You tried hitting the URL with `curl` from the pipeline instead, and it returned a login page: the export endpoint needs the session that lives in the browser, not in your shell.

Solution

```
Scenario: Programmatic download from URL
  Given I am on "/reports"
  When I download the file from the URL "/api/exports/q1-summary.pdf"
  Then the downloaded file name should be "q1-summary.pdf"
```

This is the part teams miss: the URL variant fires the request *from inside the browser context*, so the session cookies, auth headers, and anything else the page carries ride along automatically. A `curl` or test-runner-side fetch would arrive anonymous and get redirected to login. The relative path resolves against your `launchUrl`, the `LAUNCH_URL` environment variable, `http://localhost:8080` by default, so the step doesn't care which environment it's pointed at. The response is captured as a download exactly as in Recipe 387, name assertion and all.

At scale: the relative-path-plus-`LAUNCH_URL` combination is what makes this step portable across a portfolio. The same scenario file runs against dev, staging, and production because the base URL is wired per pipeline as an environment variable, nobody edits a config file per project. If you catch yourself hardcoding `https://staging.example.com/api/...` in a download step, stop: that scenario just became single-environment.

Recipe 390 Asserting Downloaded File Contents

Problem

The download works. The name is right, the size looks plausible, the test is green, and the totals inside are from the wrong quarter, because a caching bug serves last period's report under this period's name. A name assertion can't see inside the file. When the file *is* the deliverable, a sales report someone forwards to a director, you need to assert what's written in it, not just that it arrived.

Solution

```
Scenario: Report contains the expected totals
  Given I am on "/reports/sales"
  When I download the file from the link "Download text report"
  Then the downloaded file should contain:
    """
    Total revenue: $12,450.00
    Total orders: 87
    """
```

The step reads the captured temp file as text and runs a substring match against it. One detail worth knowing: the docstring is a single string, newlines included, so those two lines must appear in the file *together, in that order, adjacent*. If the real report prints other lines between them, split into two separate content assertions instead of one block. This works for any text-based format, plain text, CSV, JSON, XML. It does not work for binary formats: grepping a PDF's raw bytes will match garbage or nothing, and a ZIP needs the archive-aware steps below.

Pro Tip: for CSV exports, assert the header row. It's the schema contract, column names and order, and it breaks the moment someone reorders or renames a column, which is exactly the change that silently corrupts the spreadsheet your finance team imports. For PDFs, extract text in a hook (any PDF-to-text library) and assert against the extracted string, never the raw file.

Recipe 391 Asserting ZIP Archive Contents

Problem

Your “Download project” button packages a working directory into a ZIP. Six months in, someone refactors the packaging script and a whole folder quietly drops out of the archive. The suite stays green, it only ever checked that *a* ZIP downloaded, and the first report comes from a customer who unzipped an incomplete project. Archives are a blind spot: the test verified the envelope and never opened it.

Solution

```
  Scenario: Project ZIP has the expected files
    Given I am on "/projects/42"
    When I download the file from the link "Download project"
    Then the downloaded file should be a zip archive containing the
following files named:
      | README.md         |
      | src/index.js      |
      | tests/index.test  |
```

The step reads the captured ZIP in memory, nothing is extracted to disk, and walks the archive’s entry list, checking that every row in the table exists as an entry path. Matching is exact: case-sensitive, full path from the archive root, forward slashes (ZIP entries use `/` regardless of the OS that built them). A missing entry fails with the name of the file that wasn’t there, which is precisely the message you want at triage time.

At scale: treat the table as your packaging manifest. It is the written contract of what ships, and it lives in a feature file a product owner can actually read. Keep it next to the packaging code in the repo so the developer who adds a file to the export updates the expected list in the same pull request, review catches the mismatch, not the customer.

Recipe 392 ZIP Contents by Partial Name

Problem

Your nightly backup ZIP stamps every entry with a timestamp: `db-snapshot-2026-06-11-0300.sql.gz`. You can't pin exact names without rewriting the test every night, but you absolutely need to know the snapshot is in there, the day a refactor renames the prefix from `db-snapshot-` to `database-`, your restore runbook stops matching reality, and you find out during an actual restore. That is the single worst moment to learn your backup format changed.

Solution

```
  Scenario: Backup ZIP contains a snapshot
    Given I am on "/admin/backups"
    When I download the file from the link "Download latest backup"
    Then the downloaded file should be a zip archive containing the
following files partially named:
      | db-snapshot-      |
      | media-archive-    |
```

The `partially named` variant runs each table row as a substring match across every entry path in the archive; the assertion passes when at least one entry matches each row. Same in-memory unpacking as Recipe 391, looser comparison, it's the archive version of the filename rule from Recipe 388: assert the stable prefix, never the generated token. Mind that "substring anywhere in the path" is genuinely anywhere: `db-snapshot-` also matches `old/db-snapshot-x`. Stack this step with the strict `named` variant in the same scenario when you want both guarantees, "README.md exists exactly, and something matching `db-snapshot-` exists too."

Recipe 393 Negative ZIP Assertion

Problem

Here is a failure mode that ends up in incident reports: a developer adds a `secrets.env` to the build directory for local convenience, the export packager globs the whole directory, and the next public ZIP ships credentials to anyone with the download link. Nobody notices for months because every test asks "is the right stuff in the archive?" and none asks "is the wrong stuff out of it?". The positive assertions in Recipes 391-392 cannot catch this class of bug by design.

Solution

```
  Scenario: Public export excludes secrets
    Given I am on "/projects/42"
    When I download the file from the link "Download public project"
    Then the downloaded file should be a zip archive not containing the
following files partially named:
      | secrets.env       |
      | private/          |
      | .git/             |
```

The negation walks *every* entry path in the archive and fails the moment any listed substring appears in any of them. Directory rows like `private/` work because ZIP entries carry their full path, `private/keys/prod.pem` contains the substring and trips the guard. The failure message names the offending entry, so triage is instant. Note the asymmetry with the positive step: `containing` needs one match per row to pass; `not containing` needs zero matches across the whole archive.

At scale: make this a standing security smoke test on every public artifact in the portfolio, one shared scenario template, the same forbidden list (`secrets`, `.env`, `.git/`, `private/`, `node_modules/`), tagged `@security` so the nightly farm runs it everywhere. It costs seconds

per project. The failure mode it prevents costs a disclosure notice. Maintain the forbidden list centrally with whoever owns security review, not per project, so a new pattern (say, `id_rsa`) lands in every suite at once.

Recipe 394 Recording Video of a Test Run

Problem

The booking flow fails once a week in CI and never on anyone's machine. The failure screenshot shows the end state, a page without a confirmation number, but not the *path* there: did the search return nothing, did the Reserve click land on the wrong row, did a modal eat the Confirm press? It's the Tuesday before release, the triage thread is twelve messages deep, and everyone is guessing. A 30-second video would end the argument in one viewing.

Solution

```
Feature: Booking flow

  Scenario: Reservation completes end-to-end
    Given I start video recording
    And I am on "/booking"
    When I fill in "dates" with "2026-06-01 to 2026-06-05"
    And I press "Search"
    And I click "Reserve" in the "Sunset Suite" row
    And I press "Confirm"
    Then ".confirmation-number" should be visible within 10 seconds
    And I stop video recording
```

There are two ways to record in webship-js, and knowing which to reach for matters.

The first is the `WEBSHIP_VIDEO` environment variable, with four modes: `off`, `on`, `on-failure`, and `tag`. For the problem above, “show me every failure”, `on-failure` is the right answer, set once in CI with zero Gherkin changes:

```
WEBSHIP_VIDEO=on-failure npx cucumber-js
```

Videos land in `WEBSHIP_VIDEO_DIR` (default `./videos`) at 1280x720, named by the pattern `{datetime}.{feature_file}.{scenario}.{status}.{ext}`, note `{status}` in the name, which is what lets your artifact step glob only the failures. In `tag` mode, only scenarios tagged `@video` record; in any mode, `@no-video` suppresses recording for a scenario and `@video` forces it on.

The second way is the in-scenario steps shown above, and they come with a warning that belongs in bold. Playwright can only attach a recorder when a browser *context* is created, so `start video recording` and `stop video recording` rebuild the context mid-scenario. Rebuilding the context throws away everything in it: cookies, session, localStorage, the page you were on. Drop `Given I start video recording` after a login step and you will continue the scenario logged out, on a blank page, wondering why. That is exactly why this example puts it on the *first* line, before any state exists worth losing. Use the steps when you want recording scoped inside one scenario; use the env var for everything else.

At scale: set `WEBSHIP_VIDEO=on-failure` once in the shared CI configuration, not per project, it’s a pipeline concern, and it belongs next to the other env wiring (`LAUNCH_URL`, browser choice). Budget the storage: forty suites recording nightly add up even at 1280x720 webm, so let the `{status}` token do the work, upload only `*.failed.*` as artifacts and purge the rest at the end of the job. A green run’s video is a file nobody will ever watch.

Recipe 395 Saving Video With a Custom Name

Problem

You've finally reproduced ticket #1284, the checkout that misbehaves when "Apply" is clicked with an empty coupon field, and you want to attach the recording to the ticket. But the artifacts directory holds forty videos named by the default datetime pattern, and "which timestamped file is the repro?" is a question you'll be asked three times this week. A recording you made on purpose deserves a name you chose on purpose.

Solution

```
Scenario: Repro for ticket #1284
  Given I start video recording
  And I am on "/checkout"
  When I click "Apply"
  And I save the current video as "bug-1284-empty-coupon.webm"
```

(The coupon field is deliberately never filled, the empty submit *is* the bug being recorded.) The save step takes the recording in progress and writes it under the exact filename you give it, instead of the default `{datetime}.{feature_file}.{scenario}.{status}.{ext}` pattern. Because the name is stable across runs, re-running the repro overwrites the same file, the artifact link you pasted into the ticket keeps pointing at the latest recording. Remember the context-rebuild rule from Recipe 394 still applies: `start video recording` is the first step here for a reason.

Pro Tip: put the ticket ID in the filename, always. Six weeks from now, nobody searches artifacts by date, they search by ticket. `bug-1284-*.webm` is findable; `2026-06-09T03-14-22.checkout.feature` is archaeology.

Recipe 396 Printing the Video Path

Problem

First week with the suite, first local recording, and then the hunt begins. The run finished, the video exists *somewhere*, and the generated filename is a long datetime-feature-scenario string you'd have to reconstruct by hand. Watching a new teammate `ls -lt` their way through directories to find a file the framework just wrote is a small, daily kind of friction. Ask the framework instead.

Solution

```
Scenario: Locate the video
  Given I start video recording
  And I am on "/"
  When I press "Search"
  Then print video path
```

The step resolves where the current recording will be written, `WEBSHIP_VIDEO_DIR` (default `./videos`) plus the generated filename, and logs the absolute path into the test output, right in the step stream where you're already looking. No directory spelunking, no reconstructing the `{datetime}.{feature_file}.{scenario}.{status}.{ext}` pattern in your head. It's the video sibling of `Then print current URL`: a debug step you drop in while investigating and delete before commit.

Pro Tip: no video player on your CI-image-matched dev container? Drag the `.webm` straight into a browser tab, every Chromium and Firefox build plays webm natively.

Practical Tips

Downloads:

- Always assert the file name first, it's cheap and catches the "wrong file" class of bug.
- Use content assertions for text files and ZIP-content assertions for archives. Don't try to grep binary PDFs.
- For PDF content checks, render the PDF to text with a hook and assert against the text.

Video:

- Recording adds 5-10% overhead. Don't record every test by default, set `WEBSHIP_VIDEO=on-failure` in CI and let the framework keep video only where it earns its disk space. Use `@video` to force recording on a scenario you're watching, `@no-video` to silence a noisy one.
- The filename pattern includes `{status}`, so your artifact-upload step can glob `*.failed.*` from `WEBSHIP_VIDEO_DIR` and skip green-run videos entirely.
- Never put `start video recording` mid-scenario after state you care about, the context rebuild logs you out and blanks the page. First step or env var; those are the two safe positions.
- Combine with screenshots at key milestones for a hybrid evidence trail: stills for the report, video for the argument-settler.

Downloads test the file the user gets. Video records the path they took. Together, they cover both the artifact and the journey.

Chapter 41: XML and YAML Response Testing

Modern APIs default to JSON. But the industry runs on more than modern APIs. SOAP services power banking, telco, and government integrations. RSS and Atom feeds drive news aggregators. Sitemaps are XML. Kubernetes manifests, Helm charts, and CI configs are YAML. Configuration files for thousands of internal tools are YAML. If your test suite handles only JSON, you have a blind spot wherever a partner, vendor, or platform speaks anything else.

Webship-js ~2.0 adds **XML response testing** and **YAML response testing** families. You parse the response, then assert against elements, attributes, namespaces, types, and structural constraints, without ever writing a custom step.

This chapter starts with XML, then moves to YAML. The two families share most concepts, if you understand one, the other is a re-read.

XML and YAML, Side by Side

CONCERN	XML STEP PREFIX	YAML STEP PREFIX
Element exists	`the XML element "..."`	`the YAML element "..."`
Element value	`... should be equal to`	`... should be equal to`
Attribute	`the XML attribute "..." on element "..."`	`the YAML attribute ...`
Namespace	`the XML should use the namespace`	`the YAML should use the namespace`
Type	(XML schema)	`the YAML value at "..." should be of type`
Multi-document	,	`the YAML response should have N document(s)`

XPath is the path language for XML. YAMLPath (`a.b[0].c`) is the path language for YAML. Both are exposed via the same step shapes.

Recipe 397 Asserting an XML Element Exists

Problem

The third client your team onboards is a logistics company, and their order API speaks SOAP, XML envelopes, not the JSON your whole suite is built around. The partner's documentation swears `<orderId>` is always in the response, but their Tuesday deploys have broken that prom-

ise twice, and each time your order pipeline failed downstream where it was hardest to trace. You need the integration test that catches it at the boundary, without writing a custom XML parser for one partner.

Solution

```
Feature: SOAP order placement

  Scenario: Order envelope contains orderId
    Given the API base URL is "https://partner.example.com/soap"
    When I send a POST request to "/orders" with body:
      """
      <?xml version="1.0"?>
      <Envelope><Body><PlaceOrder><Item>SKU-42</Item></PlaceOrder></Body>
</Envelope>
      """
    Then the response should be in XML format
    And the XML element "//orderId" should exist
```

The plumbing is the same API family you know from Chapter 11, `the API base URL is` sets the host, `with body:` sends the docstring verbatim as the request payload. What changes is the assertion layer: `should be in XML format` parses the response body into an XML document (and fails loud, with the parser error, if the partner returns an HTML error page wearing an XML content type, a thing partners do). The `exist` step then runs the XPath against that parsed document and passes if the match set is non-empty. `//orderId` means “an `<orderId>` element at any depth”, forgiving of envelope restructuring. Swap in a strict path like `/Envelope/Body/Response/orderId` when the position itself is the contract. One trap worth knowing before it costs you an afternoon: if the partner namespaces every element, a bare `//orderId` may match nothing, the portable XPath form is `//*[local-name()='orderId']`, and Recipe 401 shows how to assert the namespace itself.

At scale: SOAP partners multiply quietly across an enterprise portfolio, banking, telco, government, logistics, and the worst outcome is each project growing its own ad-hoc XML parsing. With one shared step family, the QA engineer who wrote the SOAP suite for project three reads project

nine's suite without a glossary. Keep one feature file per partner contract and tag it with the partner name, so when "the partner deployed something" alerts fire, you can run exactly that contract suite on demand.

Recipe 398 Asserting XML Element Value

Problem

Your order code treats any status it doesn't recognise as a failure, a sane default. Then the partner ships a minor release that changes `<status>Accepted</status>` to `<status>ACCEPTED</status>`, every order starts "failing" at 2am, and the on-call engineer spends an hour proving the orders actually went through. Existence checks can't catch this; the element was there all along. The *value* is the contract.

Solution

```
  Scenario: Status is Accepted
    Given the API base URL is "https://partner.example.com/soap"
    When I send a POST request to "/orders" with body:
      """
      <?xml version="1.0"?>
      <Envelope><Body><PlaceOrder><Item>SKU-42</Item></PlaceOrder></Body>
</Envelope>
      """
    Then the XML element "//status" should be equal to "Accepted"
```

The step resolves the XPath, takes the first matching element's text content, trims surrounding whitespace, and compares it to the expected string, exact and case-sensitive, which is exactly what you want for an enum-like status field, because `ACCEPTED` vs `Accepted` is precisely the bug this test exists to catch. The whitespace trim handles pretty-printed XML, where the text node is

really `\n Accepted\n`. Use `should contain` when the element wraps prose and you only care about a fragment. One thing XML doesn't give you: types. Without a schema, `<count>3</count>` is the *string* "3", compare it as one.

Pro Tip: put `Then the API response code should be 200` before the value assertion. When the partner is down, you want the failure to read "got 503", transport problem, not "expected Accepted, found nothing", which sends the triager hunting a content bug that doesn't exist.

Recipe 399 Asserting XML Attribute Value

Problem

The partner's `<order>` element carries a `currency="USD"` attribute, and their parser quietly defaults to EUR when the attribute is missing or misspelled. You learned this from an invoice that was off by exactly one exchange rate. Element-level assertions look straight past attributes, so this contract needs its own check. And you'd rather verify your assertion logic against a saved sample than depend on the partner's sandbox being up during every CI run.

Solution

```
  Scenario: Order currency is USD
    Given the response content from the file "fixtures/order.xml"
    Then the XML attribute "currency" on element "//order" should be equal
to "USD"
```

Two things are happening here. First, `the response content from the file` loads a saved XML sample from disk and treats it as if it were the live response, so the rest of the assertion family works identically against fixtures and against real calls. That decouples your contract tests from partner uptime: the live smoke test runs once, the fixture-based assertions run on every commit. Second, the attribute step takes two arguments, the attribute name and the XPath of the element

that owns it, finds the first matching element, and compares that element's attribute value. `should not be equal to`, `should contain`, and `should not contain` complete the family.

At scale: check the `fixtures/` directory into the repo and treat it as the partner's living archive. Every time a partner sends a "here's our new response format" sample, or breaks you in production, the offending document goes into fixtures and becomes a permanent regression test. When three projects integrate the same partner, share one fixtures package between them; the partner has one contract, your portfolio should have one copy of it.

Recipe 400 Asserting XML Element Count

Problem

A deduplication bug doubles every line item in the cart export, and not a single existence or value assertion notices, `//cart/item` still exists, the first item's value is still right. The customer notices, though, because they were invoiced twice per item. "The element is there" and "the element is there *exactly three times*" are different contracts, and list-shaped data needs the second one.

Solution

```
Scenario: Cart has three items
  Given the response content is the following:
    """
    <cart>
      <item id="1"/>
      <item id="2"/>
      <item id="3"/>
    </cart>
    """
  Then the XML element "//cart/item" should have 3 element(s)
```

The step evaluates the XPath, collects the full match set instead of stopping at the first hit, and asserts the set's size, exact equality, so both the dedup bug (6 items) and the lost-row bug (2 items) fail it. This recipe also shows the third way to feed the XML family: `the response content is the following:` takes the document inline as a docstring, which keeps tiny examples self-contained, handy for the assertion-logic tests themselves and for teaching scenarios like this one. Live response, fixture file, inline docstring: same assertions over all three.

Recipe 401 Asserting XML Namespace Use

Problem

Your service's SOAP responses are consumed by a strict .NET client at the partner's end, the kind that validates the envelope namespace and rejects anything else with an opaque fault. Then someone upgrades the XML library on your side, the default namespace prefix changes, and a well-meaning "cleanup" drops the declaration entirely. Browsers and lenient parsers shrug; the partner's gateway returns errors, and the integration is down while two teams blame each other's logs. Namespace compliance is invisible until the strictest consumer in the chain enforces it.

Solution

```
  Scenario: SOAP envelope declares standard namespace
    Given the response content from the file "fixtures/soap-response.xml"
    Then the XML should use the namespace
"http://schemas.xmlsoap.org/soap/envelope/"
```

The step scans the parsed document for `xmlns` declarations and passes if any of them declares the given URI. The match is on the *URI*, not the prefix, and that's the correct semantics, because `soap:Envelope`, `s:Envelope`, and `SOAP-ENV:Envelope` are all the same document as long as the prefix binds to the same URI. Prefixes are arbitrary local names; the URI is the identity.

That's also why the library-upgrade scenario above is so sneaky: the prefix changing is harmless, the URI disappearing is the outage. `should not use` inverts the check, use it to assert a legacy or internal namespace has *not* crept into a public document.

Pro Tip: when a namespace assertion fails, diff the fixture against the live response before touching code. Nine times out of ten the declaration moved (from the envelope to the body element, say) rather than vanished, still a contract change worth flagging to the partner, but a different conversation than "we broke SOAP."

Recipe 402 Asserting a YAML Element Exists

Problem

Your platform team built an endpoint that renders pod manifests for the deploy tooling, generated YAML, assembled from templates and overrides. Generated is the key word: one template refactor can drop the whole `metadata` section, the endpoint still returns 200 with perfectly valid YAML, and the deploy tool crashes at apply time. Which, by Murphy's law, is during the release window. The cheapest insurance is a test that asks the endpoint for a manifest and checks the load-bearing sections are actually in it.

Solution

```
Feature: Manifest endpoint

  Scenario: Pod manifest has metadata
    Given the API base URL is "https://api.example.com"
    When I send a GET request to "/k8s/pods/payment-api"
    Then the response should be in YAML format
    And the YAML element "metadata" should exist
```

`should be in YAML format` parses the body first and fails with the parser's error, line and column, if the response isn't valid YAML at all. That's your fail-fast layer: a tab character or a stray template artifact gets reported as "invalid YAML at line 12", not as a confusing path-not-found further down. After parsing, the document is a plain object tree, and `should exist` resolves the YAMLPath against it: dotted keys for maps, brackets for array indices, so `spec.containers[0].image` reads "the image of the first container under spec". Exists means the path resolves to *something*, including `null` or an empty map, so pair it with value or type assertions (Recipes 403-404) when "present" isn't enough.

At scale: the same assertions don't have to wait for a live endpoint. Render your Helm charts or kustomize overlays in CI, point the file-based YAML steps at the output, and every one of your twenty services gets manifest checks *before* anything reaches a cluster. The QA suite and the platform team end up sharing one vocabulary for "the manifest is well-formed", and that conversation otherwise tends to happen in an incident channel.

Recipe 403 Asserting YAML Element Value

Problem

The payment service was bootstrapped the way every service is bootstrapped: copy the billing service's manifests, rename what you remember to rename. Someone missed `metadata.name`. The YAML is valid, the endpoint returns 200, the deploy goes green, and the tooling just rolled payment code out under the billing service's name, on top of the billing deployment. Existence checks (Recipe 402) can't see this; the field is there. It's just wrong.

Solution

```
Scenario: Pod name is payment-api
  Given the YAML response content from the file "fixtures/pod.yaml"
  Then the YAML element "metadata.name" should be equal to "payment-api"
```

Same shape as XML, different path language. `the YAML response content from the file` parses the fixture once into a plain object tree; the assertion then walks the YAMLPath one dotted segment at a time, map key `metadata`, then map key `name`, and compares the scalar it lands on against the expected string. Where XPath queries a document and can return many matches, YAMLPath walks exactly one route to exactly one value, which makes failures unambiguous: either the path didn't resolve, or the value didn't match, and the message tells you which. Both families share the same comparison set, `should be equal to`, `should not be equal to`, `should contain`, `should not contain`, so the muscle memory transfers.

At scale: copy-paste bootstrapping is exactly how a twenty-service portfolio gets born, which means copy-paste residue is a portfolio-wide failure class, not a one-off. Keep a tiny "identity check" scenario, name, namespace, team label, in every service's suite, generated from the same template with one value changed. It is the cheapest test you will ever write, and it is aimed at the single most common manifest mistake in multi-project teams.

Recipe 404 Asserting YAML Value Type

Problem

A templating change wraps a value in quotes, `replicas: "3"` instead of `replicas: 3`, and nothing complains until the deploy tool does, at apply time, during the release window. This is YAML's signature trap: the difference between a number and a string is two invisible characters, every value assertion passes either way ("3" equals "3"), and the type only matters to the strictest consumer downstream. You want the type checked where it's cheap: in the suite, on every commit.

Solution

```
  Scenario: Replicas is a number
    Given the YAML response content from the file
"fixtures/deployment.yaml"
    Then the YAML value at "spec.replicas" should be of type "number"
```

Here is what's really being tested: YAML's own typing rules decided what landed in the parsed tree. An unquoted `3` parses as a number; a quoted `"3"` parses as a string; `true`, `false`, `null`, and `~` parse as their own types. The step resolves the path and applies JavaScript's `typeof` semantics to whatever the parser produced, with `Array.isArray` distinguishing arrays from objects. Supported types: `string`, `number`, `boolean`, `array`, `object`, `null`. So a failure here isn't pedantry, it means the document's quoting changed, which usually means a template changed, which is precisely the regression you wanted flagged before the cluster saw it.

Pro Tip: the most famous version of this bug is the Norway problem, in YAML 1.1-style parsers, an unquoted `country: NO` parses as boolean `false`. Put a `should be of type "string"` assertion on any field whose values come from a country, locale, or version list (`version: 1.20` parses as the number `1.2`). One line of Gherkin, and an entire category of "works for every country except Norway" bugs is gone.

Recipe 405 Asserting YAML Numeric Range

Problem

Someone load-tested on a Friday and left `replicas: 40` in the values file. Nothing is invalid, the cluster will cheerfully schedule forty pods, and the first signal anyone gets is the infrastructure bill, three weeks later, routed to a manager who has to ask "what changed?" The opposite direction is worse: `replicas: 0` is also valid YAML, and it serves exactly zero requests. Valid-but-absurd values are a class of bug that no schema and no parser will ever catch for you, because only your team knows what "sane" means here.

Solution

```
  Scenario: Replicas in safe range
    Given the YAML response content from the file
"fixtures/deployment.yaml"
    Then the YAML value at "spec.replicas" should be greater than or equal
to 1
    And the YAML value at "spec.replicas" should be less than or equal to
10
```

The step resolves the path, requires the resolved value to be numeric, then applies the comparison. Stacking two assertions, as here, gives you a closed range with two readable failure messages, or use the single `between X and Y` form when one line reads better. The full family: `greater than`, `greater than or equal to`, `less than`, `less than or equal to`, `between X and Y`. Floats are accepted. One useful side effect: if the value is the *string* `"3"`, the numeric assertion fails, which is correct behaviour, because it just surfaced the quoting bug from Recipe 404 for free.

At scale: range checks are how tribal knowledge becomes executable policy. Every platform team has unwritten rules, “never more than four replicas in dev”, “memory limits under 2Gi outside prod”, that live in people’s heads until a values file violates them. Write the guardrails once as a shared feature file, render every service’s manifests in CI, and run the same file against all of them. The violation then shows up in a pull request review instead of on the cloud invoice, and the rule survives the person who made it leaving the team.

Recipe 406 Asserting YAML Array Contents

Problem

Arrays are where path-based assertions get brittle. The `app` container you care about sits at a different index in every service, index 0 here, index 1 where a proxy sidecar comes first, so `spec.containers[0].name` is a guess dressed up as a contract, and it breaks the day someone

reorders the list for readability. Worse is the inverse case: during an incident, someone attached a debug sidecar to the pod spec and forgot to remove it. Both problems need assertions about *membership*, not position.

Solution

```
  Scenario: Containers list has the app container
    Given the YAML response content from the file "fixtures/pod.yaml"
    Then the YAML array at "spec.containers" should contain an item where
"name" is "app"
```

The predicate iterates every entry in the array, reads the named subfield (`name`) on each, and passes on the first match, order-independent by design, because the order of `spec.containers` is an implementation detail nobody promised you. That is the whole argument against index paths: `containers[0]` encodes an ordering, this step encodes an identity. The negation, `should contain no item where "name" is "debug-shell"`, walks the *entire* array and fails on any match, that's your forbidden-config guard, the YAML cousin of the ZIP secrets check from Recipe 393. Stack both in one scenario when the contract is "app must be there, debug must not."

Recipe 407 Asserting YAML Has Exactly the Expected Keys

Problem

Extra keys are how config rot starts. A field gets renamed, `replicaCount` becomes `replicas`, and the old key stays behind, silently ignored by the tooling but faithfully copied into every new service bootstrapped from this file. Eighteen months later there are nine keys at the top level, nobody can say which four are load-bearing, and the new hire's first PR edits the dead one. Presence checks can't stop this; only an *exact* key list can, because rot is made of keys that shouldn't be there.

Solution

```
  Scenario: Top-level keys are exact
    Given the YAML response content from the file
"fixtures/deployment.yaml"
    Then the YAML keys at "" should be exactly
"apiVersion,kind,metadata,spec"
```

The path `""` refers to the document root; any other YAMLPath works too, so you can pin the keys of `metadata` or `spec` just as tightly. The step collects the keys of the map at that path and compares them as a set against the comma-separated list, it fails on a missing key *and* on an extra one, naming the difference in the message, which turns triage into a one-glance job. Choose your strictness deliberately: `should be exactly` for formats you own, where every new key should be a conscious, reviewed decision; `should have keys` for "must contain at least these" semantics, when a tool you don't control is allowed to add fields around yours. Exact-keys assertions are the lint rule that makes cleanup permanent, once the dead key is removed, it can never quietly come back.

Recipe 408 Asserting YAML Document Count and No Duplicates

Problem

Your stack file holds three documents, a Service, a Deployment, an HPA, separated by `---`. Then a messy merge-conflict resolution deletes one separator, two documents fuse into one, and the apply creates two objects instead of three. The YAML still parses, so nothing upstream complains. The other trap is quieter still: the same key pasted twice into one document. Most parsers shrug and keep the last occurrence, which means the first half of your duplicated config simply stops applying, and there is no error anywhere to tell you.

Solution

```
Scenario: Multi-document manifest is well-formed
  Given the YAML response content from the file "fixtures/stack.yaml"
  Then the YAML response should have 3 document(s)
  And the YAML should have no duplicate keys
```

The parser splits the stream on `---` separators and the count assertion checks how many documents came out, so both failure directions are covered: a lost separator fuses documents and the count drops; a stray separator splits one and the count rises. Either way the number changes, and the number is the contract. The duplicate-keys check then re-examines each document with duplicate detection switched on, because default last-wins parsing is exactly what makes this bug invisible at runtime. Together the two lines are a structural lint for any hand-edited or template-generated multi-document file, the well-formedness floor under all the value assertions above.

At scale: these two steps make a good universal pre-flight. Every rendered manifest bundle in the portfolio, whatever the service, can run “expected document count, no duplicate keys” before any service-specific assertion, one shared scenario outline, parameterized only by file and count. When a platform-wide template change breaks rendering, twenty pre-flights fail with the same clear message at the same moment, which tells the on-call engineer “template problem, one fix” instead of twenty mysterious service-level failures.

Bonus: JSON Schema for YAML

Problem

Path-by-path assertions scale to a point. By the time the manifest contract covers forty fields, the feature file is longer than the manifest itself, and it still can’t say “and nothing else changed shape.” Meanwhile the schema describing this format already exists: your platform team maintains

one, or the upstream tool publishes one. Re-transcribing it into Gherkin, one assertion at a time, is the kind of duplication that drifts the first month nobody maintains it.

Solution

```
  Scenario: Pod manifest matches schema
    Given the YAML response content from the file "fixtures/pod.yaml"
    Then the YAML should match JSON Schema
"fixtures/schemas/pod.schema.json"
```

The schema is loaded from disk and the parsed YAML is validated against it, and yes, JSON Schema validates YAML just fine, because by the time the validator runs, the YAML is already an object tree and the schema never knows what text format it came from. One step asserts the whole shape: required fields, types, enums, nesting, and (if the schema sets `additionalProperties: false`) the “nothing extra” guarantee that would otherwise take a stack of exact-keys assertions. Failures report path and message, “spec.replicas must be number”, not “schema mismatch”, so a red run points at the field, not at the concept of schemas. Keep the targeted steps from Recipes 402-408 for the two or three values your team actually argues about; let the schema carry the other forty.

At scale: the schema is the contract, so have exactly one. When twelve services emit the same manifest format, publish the schema once (an internal package, or a plain shared fixtures repo) and point every suite’s step at that single file. A format change then becomes one reviewed pull request that every consuming team can see and veto, instead of twelve slowly diverging copies of the same assertions. This is also the artifact to hand the PMO when they ask what “config is valid” means: a versioned schema beats a wiki page every time.

When to Use XML/YAML Testing

You need this family when:

- Your service speaks SOAP, RSS, Atom, or XHTML
- Your CI publishes Kubernetes manifests, Helm charts, or CloudFormation YAML
- A partner API returns XML and you want to test integration without a custom parser
- A config file format is YAML and you want lint-style assertions on its structure

For pure JSON APIs, stick with the JSON assertions from Chapter 11 (`the JSON response should have property "..."`). The XML and YAML families exist for everything else.

Chapter 42: Pointer Actions, Drag and Drop, Keyboard

A `click` covers maybe sixty percent of user input. The other forty are pointer actions that look simple on the surface but require precise sequencing: drag, drop, hover, right-click, middle-click, double-click, tap, key combos. A reordering UI uses drag-and-drop. A context menu opens on right-click. A “copy to clipboard” button needs Ctrl+C to be a credible test. A mobile gesture is a `tap`, not a click.

Webship-js ~2.0 adds first-class steps for every pointer and keyboard action a real user can produce. This chapter walks each one.

Pointer Actions Map

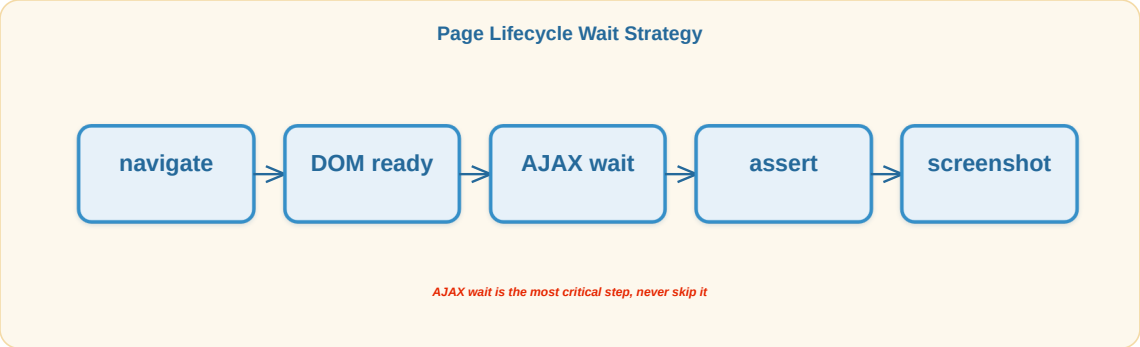


FIGURE 42.1 *Pointer Actions Map*

USER GESTURE	WEBSHIP-JS STEP
Single click	`When I click "X"`
Double click	`When I double-click on "X"`
Right click	`When I right-click on "X"`
Middle click	`When I middle-click on "X"`
Hover	`When I hover over "X"`
Tap (mobile)	`When I tap on "X"`
Drag and drop	`When I drag "A" to "B"`
Click with modifier	`When I click on "X" while holding "Shift"`
Press key	`When I press the key "Enter"`
Press chord	`When I press the keys "Control+S"`

Recipe 409 Double-Click to Edit Inline

Problem

Your spreadsheet-style grid enters edit mode on double-click, and the regression that broke it shipped clean through manual QA, because every tester single-clicked, saw the selection highlight, and moved on. Edit mode is the kind of interaction nobody re-checks once it works. Then an event-handler refactor replaces the `dblclick` listener with two `click` handlers and a home-made timer, and editing turns into a coin flip on slower machines. The support tickets say “sometimes I can’t type in cells”, which is the least debuggable sentence in the language.

Solution

```
Feature: Inline editing

  Scenario: Double-click cell enters edit mode
    Given I am on "/sheets/abc"
    When I double-click on "td[data-cell='A1']"
    Then "td[data-cell='A1'] input" should be focused within 1 second
    When I fill in the field "A1" with "100"
    And I press the key "Enter"
    Then "td[data-cell='A1']" should have text "100" within 1 second
```

The double-click fires the platform’s native `dblclick` event, two down/up pairs inside the OS double-click interval, then the `dblclick` itself, not two independent click steps, which the browser would treat as two separate selections. Apps that listen for `dblclick` (spreadsheets, tree views, map zoom) need this step rather than `click click`. The rest of the scenario rides web-ship-js’s wait machinery: `should be focused` is a web-first assertion that polls every 100ms, so the editor’s mount animation costs nothing; and because the double-click is a state-changing step, the AfterStep hook runs `smartSettle` afterwards, draining any AJAX or pending timers before the fill-in starts. The fill never races the editor mounting. `Enter` commits the value, and the final text assertion proves the cell re-rendered with it, the full edit round-trip, not just the toggle.

Pro Tip: assert focus on the *input inside the cell*, as here, not on the cell itself. “Edit mode” is really “an editable element exists and owns focus”, testing the CSS class the cell gains is testing the paint, not the behaviour.

Recipe 410 Right-Click for Context Menu

Problem

Context menus are invisible features. No button advertises them, so manual testers forget they exist, analytics undercounts them, and your file manager’s right-click menu, the one carrying “Rename” and “Delete”, can die in a refactor without anyone noticing. The classic killer is a well-meaning global click handler that swallows the `contextmenu` event for some overlay logic. Users don’t file bugs for a menu they don’t know they lost; they just call support and say the product “feels broken.”

Solution

```
Scenario: Context menu opens on right-click
  Given I am on "/files"
  When I right-click on "li[data-file='report.pdf']"
  Then ".context-menu" should be visible within 1 second
  When I click "Rename"
  Then "input[name=newName]" should be focused within 1 second
```

The step fires a real `contextmenu` event sequence, right-button down, up, then `contextmenu`, so a page handler that calls `event.preventDefault()` and renders its own menu behaves exactly as in production. One worry you can cross off: the *browser’s* native context menu never renders during automation, so there is nothing to dismiss and nothing to screenshot over your UI. From there the scenario is ordinary webship-js: a web-first visibility poll for the menu, a click on “Rename” by its visible text, and a focus assertion proving the rename input is ready for typing, which is the real user promise, not merely “a menu appeared.”

Pro Tip: add a second scenario that right-clicks, then clicks elsewhere, then asserts `".context-menu" should be hidden`. Menus that open are half the feature; menus that refuse to close are the half users actually rage about.

Recipe 411 Middle-Click to Open in New Tab

Problem

Power users read search results the same way everywhere: middle-click five of them, then work through the tabs. That convention only survives if your results are real anchors. The day a frontend "improvement" turns the link into a `div` with a click handler and router navigation, middle-click dies, and the complaint arrives phrased as "your site broke my mouse," usually from your most engaged users. You want a test that proves the link still behaves like a link.

Solution

```
  Scenario: Middle-click opens in new tab
    Given I start recording network requests
    And I am on "/search?q=playwright"
    When I middle-click on "a.search-result__link"
    Then a GET request to "**/articles/playwright-101" should have been
made
```

Middle-click is browser convention for "open in new tab/window." The step fires the native auxiliary-button event; webship-js doesn't switch tabs automatically, if you need to interact with the new tab, see the tab-switching family in Chapter 12. That's why this scenario asserts on the *network* instead of the new tab: `I start recording network requests` arms the request log before anything loads, and the final step checks that the middle-click actually produced a navigation request for the article. The `**/` glob matches the full URL whatever host it ran against, so the same scenario passes on local, staging, and production via `LAUNCH_URL`. The ordering matters, recording starts first, for the same reason the download steps in Chapter 40 arm their listener before clicking: assert-after-the-fact only works if someone was listening when the fact happened.

Recipe 412 Hover to Reveal Tooltip

Problem

The help text for the trickiest field on the form lives in exactly one place: a tooltip. Tooltips break in the most boring ways, a z-index change buries them under the header, a CSS refactor renames the class that positions them, a library upgrade changes the trigger event, and manual passes never catch it, because the testers already know what the field means and never hover. The user who actually needed the hint gets nothing, and the form's abandon rate quietly climbs.

Solution

```
Scenario: Help tooltip appears on hover
  Given I am on "/profile"
  Then "#help-tooltip" should be hidden
  When I hover over "#help-icon"
  Then "#help-tooltip" should be visible within 500 milliseconds
  And "#help-tooltip" should contain text "Your username must be unique"
```

`hover over` moves the pointer to the element's centre and leaves it there, CSS `:hover` rules and JavaScript `mouseenter` / `mouseover` handlers both fire, exactly as they would for a real pointer. Notice the scenario asserts `hidden` *first*: that proves the hover caused the reveal, rather than the tooltip having been visible (or stuck open) all along. The tightened `within 500 milliseconds` budget is deliberate. The web-first assertion polls every 100ms, and a tooltip behind a 300ms CSS transition fits comfortably, but the default 5 seconds would also pass a tooltip that takes four seconds to appear, which is its own UX bug. Timeout budgets encode expectations; spend them like money. The final `contain text` line is the content regression test, remember, this tooltip is the only home that sentence has.

At scale: tooltips almost always come from one design-system component shared across every product in the portfolio. Test the component's *mechanics*, reveal, hide, positioning, deeply in one suite, the design system's own; in the other nineteen suites, assert only the content each product

puts inside it. When the design system upgrades and the interaction breaks, one suite goes red with a precise answer instead of twenty suites going red with the same noise.

Recipe 413 Tap Gesture on Mobile

Problem

On the mobile build, the dismiss control listens for `touchstart`, not `click`, because the team wanted the snappiest possible response. Your desktop-viewport suite passes every night, and meanwhile the notification cannot be dismissed on an actual phone. This is the trap with touch UIs: a `click` step fires mouse events, and an element listening only to touch events never hears them, so the one flow your mobile users hit hardest is precisely the one your suite never exercises. The bug report eventually arrives as a screen recording from someone's commute.

Solution

```
Scenario: Notification dismisses on tap
  Given I am viewing the site on a "mobile" device
  And I am on "/"
  Then ".notification" should be visible
  When I tap on ".notification__dismiss"
  Then ".notification" should be hidden within 1 second
```

`tap` fires `touchstart` and `touchend`, the genuine touch sequence, rather than a synthesised mouse click, which is exactly what a UI listening to touch events directly needs to hear. The device step before it matters just as much: it gives the page a phone-sized viewport with touch support, the context in which a tap makes sense at all. The assertion pair tells the dismissal story, visible before, hidden within a 1-second budget after, enough headroom for a slide-out animation without tolerating a hang.

At scale: give touch-dependent scenarios their own CI lane. Tag them `@mobile` in every project and run them in a dedicated job where the device emulation is configured once, centrally, one place to decide which device profile the whole portfolio tests against. Without that convention, mobile coverage decays into whatever each project's author happened to emulate the week they wrote the test, and "works on mobile" means something different in every repo.

Recipe 414 Drag and Drop Between Lists

Problem

The kanban board *is* the product: drag a card to "Done" and the API records the move. It is also the interaction most likely to break in a frontend upgrade, because there are two incompatible ways to implement dragging, native HTML5 drag-and-drop and pointer-event libraries, and a library swap silently changes which one your page speaks. The cruel version arrives the Tuesday before release: the new board animates beautifully, the card glides across the screen, and the drop never persists. Visually perfect, functionally void, and only a test that checks *state*, not motion, will catch it.

Solution

```
  Scenario: Card moves to Done column
    Given I am on "/board/sprint-12"
    When I drag "[data-card='TASK-42']" to "[data-column='done']"
    Then "[data-column='done'] [data-card='TASK-42']" should be visible
within 2 seconds
    And "[data-column='todo'] [data-card='TASK-42']" should not be
attached
```

The drag-and-drop step is HTML5-DnD aware: it fires `dragstart`, `dragenter`, `dragover`, `dragleave`, `drop`, and `dragend` in the sequence a real drag produces. For libraries that never touch native DnD and listen to pointer events instead (dnd-kit and friends), the step still works, it dispatches pointer events as a fallback, which is exactly the both-implementations insur-

ance the library-swap problem calls for. The assertions are the both-sides pattern: the Done column *gained* the card and the To-do column *lost* it, `not be attached` checks actual DOM removal, a stronger claim than `hidden`. The 2-second budget absorbs the drop handler's animation and save call; and since the drag is a state-changing step, the AfterStep `smartSettle` drains the save's XHR (watching `window.__webshipAjaxCount` fall to zero) before the first assertion even starts polling.

At scale: the selectors here are the real portfolio decision. Agree once, across every project, that draggables carry `data-card`-style identity attributes and targets carry `data-column`, then put the names in each project's shared selectors JSON under `tests/selectors/` so feature files reference names while the raw selectors live in one file. When the next board library changes the DOM, you update one JSON file per project, not forty feature files, and the convention survives team turnover because it is written down where the tests load it.

Recipe 415 Click With a Modifier Key

Problem

Shift-click range selection is muscle memory from every file manager since the nineties, and your bulk-actions feature depends on it. It has regressed before: a selection-state refactor rebuilt the list on every click and dropped the anchor item, so Shift-clicking item three selected exactly one row. Manual QA missed it completely, clicking three items one at a time also works, and that is how testers click. Only a test that holds the modifier the way a user does can tell the two behaviours apart.

Solution

```
Scenario: Shift-click extends selection
  Given I am on "/library"
  When I click on "li[data-book='1']"
  And I click on "li[data-book='3']" while holding "Shift"
  Then ".selected-book" should have a count of 3 within 1 second
```

The `while holding` clause presses the modifier down, performs the click, and releases, so the page's click handler sees `event.shiftKey === true`, indistinguishable from the real gesture. Supported modifiers: `Shift`, `Control`, `Alt`, and `Meta` (Cmd on macOS); the key is held only for that one click, so the next step starts unmodified. The count assertion is the part that actually proves range semantics: items 1 and 3 were clicked, but the assertion demands *three* selected elements, item 2 must have been swept up by the range. `should have a count of` is web-first, polling until the selection class lands on all three or the budget runs out. Asserting "the thing I didn't click is selected" is the whole test; the two clicks are just setup.

Recipe 416 Pressing a Single Key

Problem

Your docs site copied the `/` shortcut from GitHub: press slash anywhere, land in the search box. Then a navbar refactor re-scopes the global keydown listener, the shortcut dies, and no dashboard ever notices, shortcut users don't file bugs, they grumble, reach for the mouse, and quietly stop being your fastest users. Keyboard shortcuts are features with no pixels of their own, which makes them the easiest features to break without knowing.

Solution

```
Scenario: Slash opens search
  Given I am on "/docs"
  When I press the key "/"
  Then "#search-input" should be focused within 1 second
  And "#search-input" should be visible
```

`press the key` dispatches a single keydown/keyup pair to the page with nothing focused, and that "nothing focused" is the point of the test. A global shortcut's contract is precisely that it works when the user is just *on the page*, so the keystroke must be caught by a document-level listener, not by a lucky focused element. The two assertions then close the loop: `focused` proves focus

actually moved into the search input (the user can type immediately, the real promise of a focus shortcut), and `visible` proves the input isn't focused-but-hidden, a genuinely confusing state that focus management bugs produce more often than you'd think. For component-scoped shortcuts, add `on the element "..."` to target the keystroke, that variant is Recipe 418's territory.

At scale: keep a one-feature-file shortcut inventory per product, every binding, one scenario each, tagged `@shortcuts`. It costs minutes and it doubles as living documentation: when a product owner asks "what shortcuts do we actually support?", the feature file is the answer, and unlike the wiki page it runs every night. Shortcuts are also an accessibility surface, so your a11y review can point at the same file.

Recipe 417 Pressing a Key Combination

Problem

Ctrl+S is a promise: the user's work will not vanish. Your editor wires it to a save endpoint, but one missing `event.preventDefault()` after a refactor hands the keystroke back to the browser, which helpfully offers to save the *web page* as HTML. The user dismisses the dialog, assumes their document saved, and closes the tab. Nothing in the UI looked broken; the data is simply gone. You want proof, on every run, that the chord reaches your handler and produces a write.

Solution

```
Scenario: Ctrl+S saves the document
  Given I start recording network requests
  And I am on "/docs/42/edit"
  When I fill in the field "title" with "New title"
  And I press the keys "Control+S"
  Then a POST request to "**/api/docs/42/save" should have been made
```

The keys step accepts a `+`-separated chord: `Control+S`, `Shift+Tab`, `Control+Alt+Delete`. The modifiers are pressed and held in order, the final key is pressed and released, then the modifiers release, the same event stream a human produces. The interesting choice is the assertion: the proof of saving lives on the wire, not in the UI. `I start recording network requests` arms the request log before the page loads, and the final step demands a POST matching the `**/api/docs/42/save` glob, which matches the full URL on any host, so the scenario travels across environments with `LAUNCH_URL`. Asserting the POST instead of a "Saved!" toast tests the contract (data left the browser) rather than the decoration; toasts have a way of staying green after the request beneath them broke.

At scale: chords are where "passes locally, fails in CI" gets personal. Your CI farm runs Linux and tests `Control+S`; half the team develops on macOS where the same muscle memory is `Meta+S`. Decide once, portfolio-wide, which chord the suites assert, the CI platform's, and write the decision down next to the pipeline config, or wrap the difference in one shared custom step that branches on `process.platform`. The expensive outcome is each project solving it differently, so every new hire rediscovers the mismatch one red pipeline at a time.

Recipe 418 Pressing Keys on a Specific Element

Problem

Ctrl+B must bold text only when the editor owns focus, the same chord at document level should do nothing. Scoping bugs here are maddening because they pass in development by accident: the editor *happens* to have focus when you try it, so the shortcut works. Then in production a toast notification steals focus for two seconds, the user hits Ctrl+B during that window, the keystroke lands on the document, and bold "doesn't work sometimes." Intermittent, focus-dependent, unreproducible on demand, the worst kind of ticket.

Solution

```
Scenario: Bold shortcut bolds selected text
  Given I am on "/articles/new"
  When I fill in the WYSIWYG field "content" with the "Hello world"
  And I select "Hello" text in "content" field
  And I press the keys "Control+B" on the element ".ProseMirror"
  Then ".ProseMirror strong" should have text "Hello" within 1 second
```

The `on the element` variant delivers the key events to that specific node, so the keydown originates *inside* the editor and bubbles up through it, which is what editor keymaps (ProseMirror plugins, TinyMCE shortcuts) actually listen for. Without the scope, the keystroke originates at the document, never passes through the editor's handler, and the test would fail even though the feature works, or worse, pass for the wrong reason. The preceding select-text step (its mechanics live in Chapter 43) sets up the selection the chord operates on; the final assertion checks the *semantic result*, a `strong` element containing exactly "Hello", rather than a toolbar button's active state. Markup is the contract a bold command makes; the glowing button is just its reflection.

Drag and Drop, the Robust Way

Drag-and-drop is the flakiest of all pointer actions. A few rules:

1. **Always assert before and after.** Don't assume the drag worked, check that the target list got the item AND the source list lost it.
2. **Allow a budget.** Drop handlers often run animations or save calls. Use `within N seconds` on the assertion.
3. **Use stable selectors.** `nth-child` selectors break the moment a drag changes order. Prefer `data-id` attributes that survive reordering.

```
# Bad
When I drag "li:first-child" to "li:last-child"

# Good
When I drag "[data-card='TASK-42']" to "[data-column='done']"
```

The bad version targets positions; the good version targets identity.

Common Gotchas

Hover and CSS transitions: A `:hover` style with a 300ms transition needs `within 500 ms` on the assertion, not the default 5 seconds.

Tap vs click on mobile: If you test the mobile site with `mobile` viewport but still use `click`, you'll fire mouse events instead of touch events. Pages that listen specifically for `touchstart` won't respond. Use `tap` for mobile-only flows.

Modifier keys on Mac vs Linux: `Control+S` saves on Linux/Windows, `Meta+S` saves on macOS. If your tests run cross-platform, branch on `process.platform`.

Drag-and-drop in iframes: The drag must start and end in the same frame. To drag from the parent into an iframe (or vice versa), simulate the gesture with separate pointer-down, pointer-move, and pointer-up steps, a single `drag X to Y` won't cross frame boundaries.

Pointer and keyboard actions are where a test suite goes from "covers the happy path" to "covers what users actually do." Pick the step that matches the user's gesture, not the one that's closest to "click."

Chapter 43: Advanced Field Types

A text input is easy. `fill in "X" with "Y"` covers it. The interesting fields are the ones that don't behave like text. A color picker has a hex value and a paint swatch. A datetime field has two halves. A multi-value field accepts a list of strings, one chip per value. A WYSIWYG editor is an entire embedded document with its own keystrokes. Picking these up correctly is the difference between a test suite that covers real forms and one that pretends every input is a `<input type="text">`.

Webship-js ~2.0 adds dedicated steps for each of these field types. This chapter walks them.

Field-Type Overview

FIELD TYPE	HTML	WEBSHIP-JS STEP
Color	`<input type="color">`	`fill in the color field "X" with the value "Y"`
Datetime	`<input type="datetime-local">`	`fill in the datetime field "X" with date "Y" and time "Z"`
Datetime range	start + end pair	`fill in the start datetime field ...` + `fill in the end datetime field ...`
Multi-value (chips)	tag-style input	`fill in the multi-value field "X" with the following values:`
WYSIWYG	Trix, TinyMCE, ProseMirror, etc.	`fill in the WYSIWYG field "X" with the "Y"`
Text selection	any text field	`select "X" text in "Y" field`

Recipe 419 Filling a Color Picker

Problem

White-label products live or die on the brand settings page. A tenant sets their primary color, presses Save, and if that save quietly fails, their whole product wears a stranger's identity, which a client notices faster than any functional bug. Manual testers skip this field for an honest reason: the

native color-picker popup is a fiddly OS dialog that's miserable to operate by hand. Which is exactly the argument for automating it.

Solution

```
Feature: Brand settings

  Scenario: Save brand color
    Given I am on "/settings/brand"
    When I fill in the color field "Primary color" with the value
"#358FC8"
    And I press "Save"
    Then ".color-saved-toast" should be visible within 2 seconds

  Scenario: Color value persists
    Given I am on "/settings/brand"
    Then the color field "Primary color" should have the value "#358FC8"
```

The step never opens that OS color dialog, it couldn't, reliably: the picker UI is platform-native, different on every OS, and absent in headless runs. Instead it writes the value straight into the `<input type="color">` and fires the input events a framework binding listens for. Hex values in three- and six-digit forms both work (`#fff`, `#358FC8`); browsers normalise the input's value to lowercase six-digit internally, so the assertion compares the normalised form and the matcher is case-insensitive, `#358FC8` and `#358fc8` are the same color and the same pass. The two-scenario split is the real design here: every scenario gets a fresh browser context, so when the second scenario reads the value back, nothing could have survived from the first except through the server. A green second scenario *is* the persistence proof.

At scale: in a multi-tenant portfolio, brand settings are per-client configuration and this same set-save-reload triple applies to every tenant theme. Tag it `@branding`, drive the target site per pipeline through `LAUNCH_URL`, and keep the expected hex values in per-environment data rather than hardcoded in twenty copies of the feature. The run that catches a broken brand save the week before a client's launch pays for itself in one phone call you never receive.

Recipe 420 Filling a Single Datetime Field

Problem

The first attempt at this test was a plain fill-in with `"06/15/2026 2:30 PM"`, and the field ended up empty, the form complained about a missing start time, and someone lost an hour to it. That's the datetime-local trap: the input *displays* dates in the user's locale, but it only *accepts* the ISO wire format, and a plain text fill speaks to the display. The booking flow can't be tested at all until this one field cooperates, so it blocks every scenario behind it.

Solution

```
  Scenario: Set reservation start
    Given I am on "/booking/new"
    When I fill in the datetime field "Start" with date "2026-06-15" and
time "14:30"
    And I press "Continue"
    Then ".chosen-start" should have text "Jun 15, 2026 2:30 PM" within 2
seconds
```

The step composes the two parts into the ISO wire value (`2026-06-15T14:30`) and sets it on the native input in one call, date in ISO (`YYYY-MM-DD`), time in 24-hour (`HH:MM`). Your test speaks the wire format; the browser renders whatever the user's locale dictates; the two never collide. The field itself is found label-first, "Start" matches the label, then falls back through id, name, and placeholder, the same resolution order as every webship-js fill step. Note what the assertion checks: not the input's raw value but the *rendered* confirmation, "Jun 15, 2026 2:30 PM". That closes the whole round trip, ISO in, application formatting out, in one line.

At scale: locale is a classic source of "passes on my machine, fails in CI" for date tests, because a developer's laptop renders `15/06/2026` while the CI image renders `06/15/2026`. This step makes the *input* side locale-proof, ISO is ISO everywhere. For the assertion side, either pin the CI

image's locale and document it in the shared pipeline config, or assert on values you control (the input's ISO value) instead of locale-formatted display strings. Decide once for the portfolio; per-project workarounds for locale are how forty suites end up with forty date formats.

Recipe 421 Filling Only the Date Part

Problem

Business rule: every booking ends at 18:00, so the form pre-fills the time and users only pick a day. If your test fills both halves anyway, it silently *overwrites* the pre-filled time, and now the test edits the very state it was supposed to observe. The day the prefill rule regresses and the field comes up blank, the both-halves test stays green, because it papered over the hole itself. Tests that touch only what the user touches catch what the user would catch.

Solution

```
  Scenario: Set only the date
    Given I am on "/booking/new"
    When I fill in the date part of the datetime field "Start" with "2026-
06-15"
    Then "input[name='start']" should have value "2026-06-15T18:00"
```

The "date part" variant updates only the date component of the input's value and leaves the time component exactly as it found it; a "time part" variant covers the inverse case. That makes the assertion quietly powerful: `2026-06-15T18:00`, the ISO serialisation of a datetime-local value, verifies *two* facts in one string. The date half is what your step set; the `18:00` half is what the business rule was supposed to pre-fill, untouched by the test. If the prefill breaks, this scenario fails on the time half even though every step in it succeeded. One assertion, your action and their rule, checked together.

Pro Tip: when a form mixes user-entered and rule-driven values, write the assertion against the combined result like this rather than asserting the halves separately, the combined string also catches the subtle bug where setting one half *clears* the other.

Recipe 422 Datetime Ranges

Problem

Check-in and check-out are one logical field wearing two inputs, grouped under a single "Reservation" label. Naive label resolution finds two candidates and picks one, and which one it picks is an accident of DOM order, which means the test fills check-in twice and the booking spans negative three nights. The duration between the two fields is also the number the guest pays by, so this isn't a corner to leave untested until a date-math refactor gets it wrong.

Solution

```
  Scenario: Set check-in and check-out
    Given I am on "/booking/new"
    When I fill in the start datetime field "Reservation" with date "2026-
06-15" and time "14:00"
    And I fill in the end datetime field "Reservation" with date "2026-06-
18" and time "11:00"
    And I press "Continue"
    Then ".duration" should have text "3 nights" within 1 second
```

The start/end variants encode the convention that two datetime inputs grouped under a shared label form a range: both steps locate the pair through the "Reservation" label, then "start" addresses the first input of the pair and "end" the second. The ambiguity that breaks the plain step is resolved by saying which half you mean, in words a product owner can read aloud. The assertion is deliberately not about the inputs, it's about `.duration`, the *derived* value. June 15 to June 18 is three

nights only if the application's date math is right, so "3 nights" tests the computation, which is the part of a range field that actually carries money. Each fill is a state-changing step, so the AfterStep `smartSettle` lets any recalculation finish before Continue fires.

Pro Tip: add one more scenario with end before start and assert the form refuses it. Range fields have two contracts, compute the span, and reject the impossible span, and the second one is the one that reaches production untested.

Recipe 423 Multi-Value Chip Fields

Problem

Every project picks a different tag-input library, and every library commits a chip differently: one on Enter, one on comma, one on blur. The naive approach, fill the text box with `"testing playwright cucumber-js"`, produces zero chips, or one giant chip, depending on the library's mood, and the article saves with broken taxonomy that nobody notices until the tag archive pages come up empty. The new hire writing their first content test hits this wall on day one.

Solution

```
Scenario: Add three tags
  Given I am on "/articles/new"
  When I fill in the multi-value field "Tags" with the following values:
    | testing      |
    | playwright   |
    | cucumber-js  |
  Then ".tag" should have a count of 3 within 1 second
  And ".tag" should contain text "playwright"
```

The step works the field like a user does, one value at a time: type the value, commit it (Enter first, falling back to comma for libraries that separate on commas), then *wait for the chip to render before starting the next value*. That wait is the part you'd get wrong by hand, chip libraries re-render the input after each commit, and typing into an element mid-re-render is how characters silently

vanish. The table rows are an ordered list, entered top to bottom. The assertions then split the proof: the web-first `count of 3` says the right *number* of chips materialised (catching the all-one-chip failure), and `contain text` spot-checks that a value survived intact rather than being split mid-word.

At scale: chip widgets vary more across a portfolio than any other field type, one product on Select2, one on a homegrown React chip box, one on whatever the CMS ships. One step phrasing over all of them keeps feature files identical from project to project, so a QA engineer moving between suites never relearns "how do we add tags here." When a project's library is too exotic for the Enter/comma convention, wrap the difference in a project-local custom step *with the same wording*, the vocabulary is the asset worth protecting.

Recipe 424 Filling a WYSIWYG Editor

Problem

The body field isn't an input. It's an entire embedded editor with its own document model, and depending on the project it's Trix, TinyMCE, or ProseMirror, each hiding the real text somewhere different. A plain fill-in finds either nothing or a hidden textarea the editor ignores, and here's why that hurts more than one broken test: every content scenario in the suite needs body text before it can test anything else. This one step gates dozens of tests, so when it's flaky, the whole suite has a reputation problem.

Solution

```
  Scenario: Write an article
    Given I am on "/articles/new"
    When I fill in "title" with "Hello World"
    And I fill in the WYSIWYG field "Body" with the "This is the first
paragraph of the article."
    And I press "Save"
    Then ".article-body" should contain text "first paragraph" within 3
seconds
```

The step detects which editor it's talking to by inspecting the DOM around the labelled field, then speaks that editor's dialect: for ProseMirror it sets the editable region's text, for Trix it dispatches the events Trix listens for, and for TinyMCE it writes inside the editor's iframe, because TinyMCE is literally a page within your page. That detection is what lets one phrasing serve every project. The assertion deliberately checks the *rendered article* after Save, not the editor's internal state: content has to survive the editor's serialisation, the save request, and the re-render to count as written. A 3-second budget covers the round trip; the AfterStep `smartSettle` has already drained the save's AJAX before the assertion starts polling.

At scale: this is the pattern that carried Varbase testing at Vardot, dozens of Drupal sites, every body field a rich-text editor, one step phrasing in every suite. When the editor upgraded across the distribution, the suites reported exactly which sites' content forms broke, and nobody maintained a per-project editor step to find out. If your portfolio mixes editor libraries, the single phrasing is what keeps a QA engineer's knowledge portable between projects, the editor is an implementation detail, and the feature files stop caring about it.

Recipe 425 Selecting Text Inside a Field

Problem

To test bold, you first need a selection, and a selection is invisible state with no element to click and no attribute to read. The human gesture is click-drag across exactly one word; scripting that with raw mouse coordinates means measuring where “Hello” is painted, which breaks on the next font change, zoom level, or viewport. Teams skip formatting tests for exactly this reason, and then a toolbar refactor ships bold-that-doesn’t-bold to a newsroom.

Solution

```
Scenario: Bold a single word
  Given I am on "/articles/new"
  When I fill in the WYSIWYG field "Body" with the "Hello world"
  And I select "Hello" text in "Body" field
  And I press the keys "Control+B" on the element ".ProseMirror"
  Then ".ProseMirror strong" should have text "Hello" within 1 second
```

`select X text in Y field` finds the first occurrence of the substring within the field’s value and sets the selection range to it, by text, not by coordinates, so layout, fonts, and viewport size never enter the picture. That selection becomes the target for whatever keyboard shortcut comes next; the chord is scoped onto the editor element (Recipe 418’s trick) so it bubbles from inside the editor’s keymap. The assertion closes on the markup: a `strong` element containing exactly “Hello”, not “Hello world”, which would mean the selection grabbed too much. Selection, chord, markup: three steps that together pin down a behaviour most suites silently leave untested.

Pro Tip: “first occurrence” is load-bearing. If the body says “Hello, Hello, goodbye”, the step selects the first “Hello”, when you need a later one, make the search string longer and unambiguous (“Hello, goodbye”) rather than hoping.

Recipe 426 Selecting a Character Range

Problem

Sometimes the test is about position, not words: a copy-paste flow, a masked input that reveals a slice, a “select the middle and retype” edge case a user reported. There is no substring to search for because the *position* is the point, and dragging a mouse across exactly nine characters is not something automation should ever attempt. You need to say “characters 5 through 13” and have it mean that, on every machine, at every font size.

Solution

```
Scenario: Select character range
  Given I am on "/profile"
  When I fill in "bio" with "I am a developer working on test tooling."
  And I select from 5 to 14 text in "bio" field
  And I press the keys "Control+C"
  Then the clipboard should contain "a develop"
```

The range form uses zero-indexed character positions, half-open: `from 5 to 14` selects indices 5 through 13, here, “a develop”. Half-open intervals look pedantic until you notice they match how JavaScript slices strings: `value.substring(5, 14)` returns exactly the selected text, so you can compute your expected value in a console with the same numbers the step uses, no off-by-one negotiation. The chord then copies the selection and the clipboard assertion verifies what actually landed there, which tests the full path users rely on, selection through clipboard, not just that a highlight appeared.

Pro Tip: pin the field’s value inside the scenario, as this one does with its fill step, instead of relying on fixture data. Index-based selections are only as stable as the string under them, a fixture edit that shifts one character turns “a develop” into nonsense and the failure into a puzzle.

Recipe 427 Selecting All Text

Problem

A pre-filled field is a different animal from an empty one. Rename and change-email flows are where masked inputs and controlled components fight back: typing appends to the old value instead of replacing it, or the component restores the old value on blur because it never registered a proper clear. The user's actual gesture is select-all, delete, type, and when a field misbehaves, the most reliable fix is to make the test perform exactly that gesture instead of arguing with the component's change detection.

Solution

```
Scenario: Clear by select-all and type
  Given I am on "/profile"
  Then "input[name=display-name]" should have value "Old Name"
  When I select all text in "display-name" field
  And I press the key "Delete"
  And I fill in the field "display-name" with "New Name"
  Then "input[name=display-name]" should have value "New Name"
```

`select all text in X field` is the keyboard equivalent of clicking into the field and pressing Ctrl+A, the whole value becomes the selection, and the `Delete` keypress collapses it to a genuinely empty field, with every keystroke flowing through the events a framework's change tracking listens to. Programmatic clears sometimes skip those events, which is precisely how a component "restores" a value the test thought it removed. Two details carry weight here. The opening assertion pins the precondition (`Old Name` is really there), so when fixture data changes, the test fails at the setup line with an honest message instead of at the end with a baffling one. And the closing assertion reads the input's value, a check that the replacement took, not merely that typing happened.

Recipe 428 Field State Assertions

Problem

Modern signup forms enable the Submit button off field validity, which makes “the email field is valid” load-bearing UI state that no screenshot can show you. The classic regression: someone tightens the input’s validation pattern for an edge case, every address with a plus sign in it turns invalid, and signups sag for a week while the form looks perfectly normal in every demo. The validity state changed; nothing visible did. That gap is exactly what these assertions watch.

Solution

```
Scenario: Email field validates
  Given I am on "/signup"
  Then the field "email" should be required
  And the field "email" should be empty
  When I fill in "email" with "user@example.com"
  Then the field "email" should not be empty
  And the field "email" should have "valid" state
```

The state predicates wrap HTML5 constraint validation, the browser’s own verdict on the field, the same machinery that decides whether a form submits. `required` reads the field’s constraint; `empty` / `not empty` read its value; `valid` / `invalid` ask the element’s validity state, which accounts for every constraint at once (type, pattern, required, length). The remaining states, `pristine`, `dirty`, `touched`, `untouched`, describe interaction history: whether the value has changed and whether the user has visited the field, the signals form libraries use to decide *when* to show an error. The scenario’s shape is before-and-after on purpose: required and empty before typing, non-empty and valid after, so a failure tells you which transition broke rather than just “validation is off.”

At scale: make field-state scenarios a portfolio convention, every project’s signup and checkout forms get one, so reviewers stop guessing whether client-side validation has coverage. There’s a bonus audience: these assertions read like acceptance criteria verbatim. “Email is required and

must be valid before submit enables" is a sentence a product owner wrote in the ticket and a line a QA engineer can point to in the green run, the rare case where the criteria and the test literally match, which is exactly the sign-off conversation you want.

Field-Type Pitfalls

Native vs custom date pickers: A custom calendar component is not an `<input type="date">`. It's usually a button that opens a popover with day cells. For those, use `click "15"` in the popover, the datetime steps in this chapter target the native HTML5 inputs only.

Multi-value field separators: Some chip libraries treat space as a separator; others wait for Enter. The webship-js step tries Enter first, falls back to comma. For exotic libraries, dispatch the key event manually.

WYSIWYG content escaping: When the editor body contains HTML, the `fill in the WYSIWYG field` step sets the plain text, HTML tags are typed literally, not parsed. To set HTML content, use a custom step that calls the editor's API.

Color picker on Safari/Linux: The native color picker UI varies by platform. webship-js bypasses the UI and writes the value directly to the input, consistent across browsers.

Advanced field types reward precision. Use the right step for the right widget, and your form tests stay readable as the forms get richer.

Part XI: Production Practice

Chapter 44: JavaScript Errors and Tester-Friendly Error Handling

A passing test that leaves a stack trace in the browser console is a lie. Functionally the user got through. But the next user on a slightly different browser version, or with a malformed cookie, or after a slow CDN response will hit that latent error and break. Webship-js ~2.0 added a **JavaScript errors and warnings** family that turns every test into a smoke test for console health. It also added a **tester-friendly error system** that swaps Playwright's raw exception output for messages a non-engineer can read.

This chapter covers both: how to assert console health as part of every scenario, and how the friendly error pipeline reshapes failures into something a test author can act on without a debugger.

Why Console Health Matters

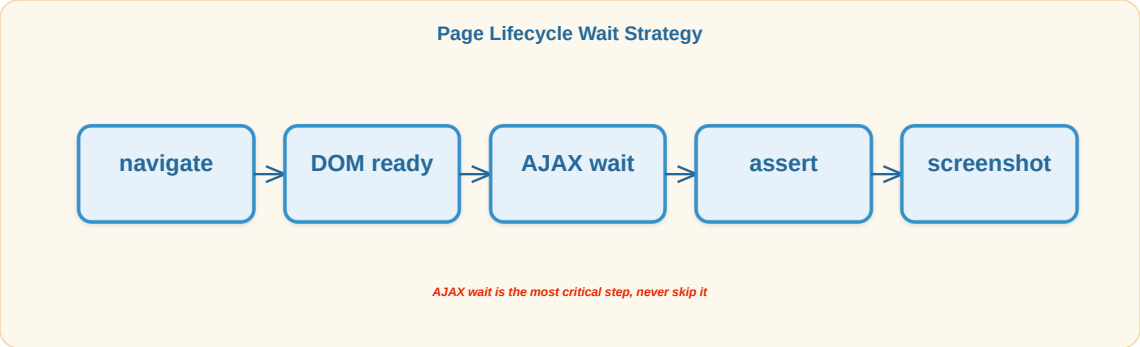


FIGURE 44.1 *Why Console Health Matters*

Browsers run JavaScript permissively. An uncaught exception in a third-party widget, a 404 from a lazy-loaded script, a deprecated API warning, none of these stop the page. They quietly log to the console. By the time a user reports “the page feels broken”, the actual cause is buried in three weeks of console noise.

A test that runs every commit can catch these the day they appear, but only if the test is wired to listen. The webship-js JavaScript-errors family installs a listener on `page.on('pageerror')` (uncaught exceptions) and `page.on('console')` (console output) for every scenario. Every event is appended to a per-scenario list in the order it fired, each entry carrying its capture type (`pageerror`, `console.error`, `console.warning`, …) plus the message text. That list is what every assertion in this chapter reads.

The whole family is controlled by a small set of environment switches, so the capture policy lives in your CI configuration instead of your feature files:

- `WEBSHIP_JS_ERROR_MODE`, `warn` (default: log to the report, don't fail), `fail` (any captured error fails the scenario), or `off`
- `WEBSHIP_JS_ERROR_LEVELS`, which console levels count; default `error`, add `warning,info` to widen the net
- `WEBSHIP_JS_ERROR_IGNORE`, a regex of known noise; matching entries are filtered out before the error checks evaluate the list
- `WEBSHIP_JS_ERROR_AFTER` (default on; set `0` to disable), run the automatic check at the end of every scenario
- `WEBSHIP_JS_ERROR_BEFORE` (off by default; set `1`), also log a pre-scenario snapshot, useful when a page survives across scenarios

Per-scenario tag overrides beat the environment: `@js-fail` (or `@javascript`) forces fail mode, `@js-warn` downgrades to warn, `@js-off` disables capture for that scenario. Keep this control panel in mind, the recipes below lean on it constantly.

Recipe 429 Asserting No JavaScript Errors

Problem

The Thursday before release, the smoke suite is green and everyone signs off. What nobody saw: the landing page has been throwing `TypeError: Cannot read property 'x' of undefined` since Monday's deploy. The assertion checked the visible headline, the headline was there, and nobody asked the console. Two weeks later the same latent error starts swallowing click handlers on a browser version you never tested, and the bug report says "the page feels broken" with zero clues.

Solution

```
Feature: Console health on landing page

  Scenario: Landing page is console-clean
    Given I am on "/"
    Then there should be no JavaScript errors
```

Run this at the end of every smoke scenario. The step is a hard assertion over the per-scenario error list that the `page.on('pageerror')` listener has been filling since the page was created. If any uncaught exception fired at any point during the scenario, not just on the final page, the step fails and the full error text lands in the report as a numbered list, so the person reading the failure sees the actual `TypeError`, not a vague “console check failed.”

Two internals worth knowing. First, this step is a hard check: it fails on captured errors even when the suite runs in the default `warn` mode. Second, it marks the list as asserted, so the automatic end-of-scenario check skips, one error is never reported twice, once by your step and once by the framework.

Note what you get even without this step: the default capture mode is `WEBSHIP_JS_ERROR_MODE=warn`, so captured errors already show up in the report as warnings. The explicit step turns that passive signal into a gate for the scenarios where console health is part of the contract.

At scale: across 20 projects, don’t paste this step into every feature file. Set `WEBSHIP_JS_ERROR_MODE=fail` once in the CI environment and let the automatic after-scenario check do the work everywhere. Exceptions then become visible tags (`@js-warn`, `@js-off`) instead of silently missing steps, and a reviewer can spot a relaxed scenario in the diff.

Recipe 430 Asserting No JavaScript Warnings

Problem

Your pages throw zero errors, so the team stopped looking at the console months ago. Meanwhile it has quietly filled with `[Deprecation] Synchronous XMLHttpRequest on the main thread is deprecated`, a countdown timer, not a footnote. The day Chrome actually removes that API, the feature breaks in production and the fix becomes an emergency patch instead of a planned ticket. You want a guard that notices the countdown while it is still cheap to act on.

Solution

```
Scenario: Landing page is warning-clean
  Given I am on "/"
  Then there should be no JavaScript warnings
```

Warnings arrive through the same `page.on('console')` listener as errors, just at a different level, and here is the catch: the level filter runs at capture time. With the default `WEBSHIP_JS_ERROR_LEVELS=error`, warning-level messages are never recorded at all, and this step would pass on a page drowning in deprecations. Pair it with `WEBSHIP_JS_ERROR_LEVELS=error,warning` so warnings actually enter the list. The step then filters the captured entries for `console.warning` and fails if any survived, which lets you keep the suite-wide policy strict about errors and use targeted scenarios like this one to police warnings where they matter.

Use warnings as a smoke check for technical debt. Deprecation warnings predict future breakage, catching them before the browser removes the deprecated API saves an emergency patch.

At scale: deprecations hurt worst at fleet scale. If 30 of your sites share the same base theme or framework version, the day a browser removes a deprecated API, all 30 break the same week. One warning-clean sentinel scenario per project, or `WEBSHIP_JS_ERROR_LEVELS=error,warning` on a nightly profile, turns that cliff into a backlog item you saw coming a year out.

Recipe 431 Matching Errors Against a Pattern

Problem

The third project you onboard arrives with a tag manager, two ad networks, and a chat widget, each of them throwing errors you will never be allowed to fix. Turn on strict console checking and every scenario fails on someone else's code; turn it off and your own bugs sail through with the noise. You need a middle path: stay silent about the rented scripts, fail loudly on anything from your own namespace.

Solution

```
Scenario: No errors from our own code
  Given I am on "/"
  Then JavaScript errors should not match "myapp\\.|MyApp"
```

The pattern is a JavaScript regex tested against the text of every recorded error. The step fails if any of them matches, so here, third-party noise is invisible to the assertion while anything mentioning `myapp.` or `MyApp` trips it.

There is a second tool for the same problem, working from the opposite end: `WEBSHIP_JS_ERROR_IGNORE` takes a regex of noise to discard, matching entries are filtered out before `there should be no JavaScript errors` and the automatic end-of-scenario check evaluate the list. Use `WEBSHIP_JS_ERROR_IGNORE` for permanent, suite-wide noise (the ad network that ships with every page) and the in-scenario negative match for per-scenario gating on first-party errors. One deliberate asymmetry: `print JavaScript errors` and the pattern step in this recipe read the raw, unfiltered list. Diagnostics stay honest, and a targeted match can still catch an error the suite has muted.

At scale: keep one `WEBSHIP_JS_ERROR_IGNORE` value in your shared CI configuration, not a per-project copy. When the team agrees that a vendor error is officially "not ours", it gets added once, with a comment and a ticket link, and every project inherits the decision. The day a project

needs its own exception, that diff is one line in one place, easy to review, easy to revert when the vendor finally fixes it.

Recipe 432 Printing Errors for Debugging

Problem

There is a test that fails roughly once a week, always with the same unhelpful timeout, never on anyone's laptop. You have stared at five failure screenshots and learned nothing, the page looks fine in all of them. Your hunch is a JavaScript error that only fires under CI timing, but by the time you open the report, the console evidence is gone. You want every run to log its full console error list so the next failure carries its own forensics.

Solution

```
Scenario: Debug captures console errors
  Given I am on "/checkout"
  When I click "Place Order"
  Then ".confirmation" should be visible within 10 seconds
  And print JavaScript errors
```

`print JavaScript errors` writes the captured list to the test output, and from there into the JSON report (`tests/reports/cucumber_report.json`), where it survives as long as your CI keeps the artefact. Each entry carries the error text plus source location, so when the weekly flake finally fires again you can diff this run's console against last week's instead of guessing.

The step is deliberately a no-op assertion, it never fails. That makes it safe to leave in place on suspect scenarios while you collect evidence across runs. Once you have identified the culprit error, replace the print with a real assertion (`JavaScript errors should not match "..."`) and the diagnosis becomes a regression guard.

Recipe 433 Combining Console Checks With Web-First Matchers

Problem

The contact form works, for now. The success banner appears, the test goes green, and a `TypeError` fires silently in the analytics callback right after submit. Nobody notices until the analytics vendor updates their script and that swallowed error starts blocking the redirect. The scenario asserted what the user sees and ignored what the browser knows. Test both.

Solution

```
Scenario: Submit succeeds AND console is clean
  Given I am on "/contact"
  When I fill in "name" with "Maria"
  And I fill in "email" with "maria@example.com"
  And I press "Send"
  Then ".success-banner" should be visible within 10 seconds
  And there should be no JavaScript errors
```

Two contracts, one scenario. The ordering here is doing quiet work. The web-first visibility check polls every 100ms for up to 10 seconds, so by the time it passes, the submit round-trip is done. Then webship-js's after-step settle kicks in, after every state-changing step, an AfterStep hook runs `smartSettle(page, 1500)`, which waits until `window.__webshipAjaxCount` is zero, pending timers are zero, and the DOM has been quiet for 250ms. That means a late error thrown from a fetch callback or a `setTimeout` has already fired and been captured by the listener before the final console assertion reads the list. You are not racing the page; the framework already waited for it.

Avoid the trap of asserting only the visible outcome, a green page with a hidden error is a latent bug with your name on the commit.

The Friendly Error System

Playwright's default error output is precise but mechanical:

```
Error: locator.click: Timeout 30000ms exceeded.
========================== logs ==========================
waiting for locator('text=Save')
==========================================================
    at /tests/step-definitions/action.steps.js:42:18
```

For an engineer, that's enough. For a tester, a product manager, or a junior team member, it's a wall. They see a stack trace and email an engineer. Multiply that across a team running dozens of suites and the failure-triage queue becomes one person's full-time job, not because the failures are hard, but because only one person can read them.

Webship-js ~2.0 ships a `friendly()` **and** `humanize()` **pipeline** that wraps every step's error in a structured, human-readable form. The same failure now reads:

```
Step failed: When I press "Save"

Could not find a button or link labelled "Save" within 30 seconds.

Hints:
  - The label may have changed. Try "Save changes" or "Submit".
  - The element may not be visible. Did you wait for the form to load?
  - The element may be inside an iframe. Did you switch to it first?

Original error: locator.click: Timeout 30000ms exceeded.
```

The tester sees the actionable summary. The original error stays at the bottom for engineers.

Recipe 434 Friendly Error in Action

Problem

A step `When I press "Save"` fails on the new hire's first PR. They open the report, find a 30-second-timeout stack trace pointing into `action.steps.js`, and conclude the framework is broken, when the actual story is that a designer renamed the button to "Save changes". You want the report to tell that story in plain English, to whoever is reading it.

Solution

You don't write anything. Webship-js wraps the step automatically. The report shows the friendly summary plus the original error. Custom steps you write benefit from the same pipeline if you wrap your error throws with `friendly()`:

```
const { friendly } = require('./webship');

When('I confirm the order in {string}', async function(button) {
  const locator = this.page.getByRole('button', { name: button });
  if (!(await locator.isVisible())) {
    throw friendly(`Could not find a confirmation button labelled
"${button}". The order may not have reached the confirmation step yet.`);
  }
  await locator.click();
});
```

`friendly()` takes three forms. `friendly(message)` makes your sentence the entire failure output, it sets the Error's `.stack` to the rendered message, which is the trick that stops cucumber-js printing a JavaScript stack trace at a tester. `friendly(message, err)` appends a `Why:` line with the original error run through `humanize()` (more on that in the next recipe). And `friendly({ action, target, hint, cause })` builds the structured form the built-in steps use: "Could not *action* '*target*'", a humanized `Why:`, and your `Hint:` line. Reach for the object form when you have a hint to offer, that is where the hints in the report come from.

The message you write matters more than the mechanism. “Could not find a confirmation button” is good; “The order may not have reached the confirmation step yet” is better, because it points at the most likely cause instead of just restating the symptom. Write the hint you would give a colleague who walked over to your desk.

Recipe 435 Humanizing Selector Errors

Problem

A custom step throws with a raw CSS selector in the message: `Could not find #cta-primary--btn`. The tester reading the report has never seen that selector, it lives three layers down in a step-definition file they have no reason to open. So the failure that should have taken thirty seconds to triage (“oh, the CTA button moved”) instead becomes a Slack thread, a screen share, and an engineer’s afternoon.

Solution

Two moves: name the element in human terms yourself, and let `humanize()` translate the low-level cause:

```
const { friendly, humanize } = require('./webship');

try {
  await locator.click();
} catch (e) {
  throw friendly({
    action: 'click',
    target: 'the CTA button',
    cause: e,
    hint: 'the button may have been renamed or moved below the fold.',
  });
}
```

`humanize()` is a translator for raw error text, not a selector lookup. It takes the original Playwright or Node error and maps known patterns to one plain-English sentence: `Timeout 30000ms exceeded` becomes “the page took too long to respond”, `strict mode violation` becomes “the locator matched more than one element”, `ECONNREFUSED` becomes “the server is not running or refused the connection”. Around twenty patterns ship with webship-js; anything unmatched falls back to a cleaned-up first line with Playwright’s noisy `Call log:` suffix stripped. Inside the object form of `friendly()`, the `cause` you pass is rendered through it as the `Why:` line, so the failure above reads: `Could not click "the CTA button". Why: the element could not be reached. Hint: the button may have been renamed or moved below the fold.`

The human name for the element is on you, and the natural source is the names you already maintain in the `tests/selectors/` JSON files for named-selector steps. If the registry calls `#cta-primary--btn` “cta button”, say “cta button” in your `target` too. One vocabulary for steps and for failures.

At scale: this is one of the strongest arguments for shared selector JSON files across projects. When 20 sites built on the same platform share a `common-selectors.json`, a failure on any of them reads the same way, “Could not find ‘main menu’” means the same thing on site 3 and site 17. A reviewer who triages failures across the whole portfolio never has to learn per-project selector dialects.

Recipe 436 Per-Scenario Hooks for JavaScript Mode

Problem

Your staging suite should scream on every console error, you own that code and you can fix it the same day. But the same suite also runs as a smoke check against production, where a marketing tag you are not allowed to remove throws on every page load. One global setting cannot serve both. The first time this hurts is the morning the production smoke run goes red over the marketing tag and someone pages the on-call engineer for a bug that is three months old and not yours.

Solution

The mode is environment-driven, with tag overrides per scenario, no custom hook required. In each pipeline, set the mode that matches what you can act on:

```
# staging pipeline: any console error fails the scenario
WEBSHIP_JS_ERROR_MODE=fail npm test

# production smoke pipeline: capture off entirely
WEBSHIP_JS_ERROR_MODE=off npm test
```

For scenario-level exceptions, use the built-in tags:

```
@js-off
Scenario: Legacy page we cannot change
  Given I am on "/legacy-widget"
  Then ".widget" should be visible

@js-fail
Scenario: Our own checkout must be console-clean
  Given I am on "/checkout"
  Then ".cart-summary" should be visible
```

Modes:

- `warn`, log captured errors to the report but don't fail (the default)
- `fail`, any captured error fails the scenario
- `off`, don't capture at all

The resolution order is fixed and worth memorising: scenario tag beats env var beats `worldParameters.javascript.mode` in `cucumber.js` beats the built-in default of `warn`. So `@js-fail` (or the legacy `@javascript`) forces fail mode for one scenario, `@js-warn` downgrades to warn, `@js-off` (or legacy `@js-errors`) disables capture, regardless of what the pipeline exported. The tags are implemented as a Before hook inside webship-js, so you get per-scenario hook behaviour without writing one. Two companion switches shape the automatic

check itself: `WEBSHIP_JS_ERROR_AFTER` (on by default; set `0` to skip the end-of-scenario evaluation) and `WEBSHIP_JS_ERROR_BEFORE` (set `1` to also log a pre-scenario snapshot, usually empty, but a lifesaver when scenarios share a long-lived page).

At scale: the env-var-plus-tags split is exactly what a multi-project CI farm wants. The farm sets `WEBSHIP_JS_ERROR_MODE=fail` once, in shared pipeline configuration; individual projects never edit it. Exceptions live as tags inside feature files, where they show up in pull-request diffs and someone has to defend them in review. The day you wonder "which scenarios have we exempted across the portfolio?", the answer is one grep for `@js-off`.

Recipe 437 Filtering Stdout in Reports

Problem

A delivery manager opens the test report before a release call, hoping for a yes/no answer. What they get is two hundred lines of Playwright internals, GPU warnings, and `Forced color profile` noise wrapped around the four lines that matter. They close the report and ask an engineer instead, which means the report failed at its one job: letting a non-engineer read the result.

Solution

The friendly error pipeline wraps `stdout` and `stderr` with a line filter that strips cucumber-js's hook-status noise, the `✔ Before # ...` and `✔ After # ...` lines that webship-js's own hooks (browser setup, settle, screenshot, video, JS-error capture) would otherwise print around every scenario in a failure dump. You don't enable it; it's on by default. A suite this instrumented runs half a dozen hooks per scenario, and without the filter a single failing step arrives wrapped in a dozen lines of green checkmarks nobody asked about. Disable it with `WEBSHIP_FILTER_HOOK_LINES=off` when you suspect a hook itself is misbehaving and you want to see every one of them.

The filter only strips hook chatter. When you genuinely need the raw firehose locally, usually when debugging a step definition, not a page, ask Playwright for its own debug channel directly:

```
DEBUG=pw:api npm test
```

That prints every Playwright API call with timings to your terminal, unfiltered, while the report stays clean. Best of both: forensic detail where the engineer is looking, readable output where everyone else is.

Recipe 438 Capturing the First Error

Problem

A broken deploy takes out one JavaScript module, and suddenly the scenario reports fourteen errors. Thirteen of them are downstream casualties, handlers that never registered, objects that never initialised. The person triaging reads all fourteen, gets overwhelmed, and files a bug describing error number nine. You want the failure message to lead with the one error that caused the rest.

Solution

Use the helper:

```
const errors = this._jsErrors || [];
if (errors.length) {
  throw friendly(`First JavaScript error: ${errors[0].message}`);
}
```

`this._jsErrors` is the per-scenario list the built-in steps read, the listeners push entries the moment events fire, each shaped `{ type, message }`, so index 0 is the first error chronologically. Uncaught exceptions (`pageerror`) and console output interleave in real firing order, and in a cascade, the first entry is the root cause far more often than not. Everything after it tends to be the page failing to cope with the first failure.

This is the same triage rule compilers taught us decades ago: fix the first error, ignore the rest, recompile. Wrapping it in `friendly()` means the report leads with that one actionable line instead of a fourteen-item wall.

Recipe 439 Asserting Specific Errors Are Allowed

Problem

Your CI runners have no outbound internet, so every run logs `Failed to load Sentry beacon`, an error that will never happen for a real user and that you cannot fix from the test side. The lazy move is to stop checking the console entirely, and that is how a real `TypeError` shipped unnoticed the last time someone made that call. You want a precise allowlist: that one error is fine, everything else still fails.

Solution

```
  Scenario: Page errors except Sentry beacon
    Given I am on "/"
    Then JavaScript errors should not match "^(?!Failed to load Sentry
beacon).*"
```

The pattern uses a negative lookahead anchored at `^`: any error that does **not** begin with the allowed text matches the pattern, and the step fails on a match. The anchor matters, without `^`, the regex engine would retry the lookahead from position 1 onwards and end up matching even allowed errors.

For multiple allowed errors, list them with alternation:

```
Then JavaScript errors should not match "^(?!(Failed to load Sentry
beacon|Cannot load Google Tag Manager)).*"
```

If an error should be allowed everywhere, not just in one scenario, promote it out of the feature file and into `WEBSHIP_JS_ERROR_IGNORE`, the suite-wide regex filter. Matching entries are dropped before `there should be no JavaScript errors` and the automatic mode check evaluate the list, so the noise stops failing scenarios everywhere at once. (The raw list still keeps them, which is why `print JavaScript errors` continues to show them, diagnostics stay unfiltered on purpose.) Scenario-level lookaheads are for local exceptions; the env var is for suite policy.

At scale: treat the ignore list like a dependency: versioned, commented, reviewed. One shared `WEBSHIP_JS_ERROR_IGNORE` value in the CI farm config, with each fragment annotated by ticket ("Sentry beacon, runners are offline, INFRA-204"), means a new project inherits years of accumulated judgment on day one instead of rediscovering each noise source the hard way.

Recipe 440 Asserting Console Cleanliness for the Whole Suite

Problem

You have 400 scenarios across 30 feature files, and the team agrees console errors should fail tests. Adding `Then there should be no JavaScript errors` to 400 scenarios is a diff nobody wants to review, and the new hire's first feature file will forget the line anyway. Policy that depends on every author remembering a step is not policy, it's hope.

Solution

Add an After hook:

```
const { After } = require('@cucumber/cucumber');
const assert = require('assert');

After({ tags: 'not @allow-console-errors' }, function() {
  const errors = this._jsErrors || [];
  if (errors.length) {
    throw new Error(`Console had ${errors.length} JavaScript
error(s):\n${errors.map(e => e.message).join('\n')}`);
  }
});
```

Every scenario now gets the check for free. Tag exceptional scenarios with `@allow-console-errors` to opt out.

Before you write this hook, know that webship-js ships the same behaviour built in: `WEBSHIP_JS_ERROR_MODE=fail` with `WEBSHIP_JS_ERROR_AFTER` left at its default `true` evaluates the captured error list at the end of every scenario automatically, and `@js-off` is the built-in opt-out tag. Zero code, same guarantee. Hand-roll the After hook only when you want something the built-in doesn't give you, a custom message format, a different tag name your suite already standardised on, or extra logic like posting the error list to a chat channel.

At scale: the built-in wins specifically because it's configuration, not code. A hook lives in one repo and gets copy-pasted (and then drifts) across the other 29; an env var lives in the shared CI template and applies to every project the moment it's set. When the policy changes, say the team decides to downgrade to `warn` during a big migration, that's a one-line change in one pipeline file, not 30 pull requests.

Recipe 441 Asserting Page Errors on Navigation

Problem

Some pages have a criminal record. `/admin/imports` lazy-loads a heavy bundle, and twice in the past year a deploy left its chunk manifest stale, both times the bug report said "the imports page is blank", and both times triage took an afternoon because no test pointed at that page. The general suite eventually went red, but with failures scattered across unrelated scenarios that all happened to pass through the broken page. You want a sentinel: one scenario that watches this exact page for this exact failure and names it the moment it returns.

Solution

```
Scenario: Imports page loads without chunk errors
  Given I am on "/admin/imports"
  Then there should be no JavaScript errors
  And JavaScript errors should not match "chunk load error"
```

The first assertion is the general gate, the error listener has been recording since the page opened, so a chunk that failed during initial load is already in the list by the time navigation settles. The second line looks redundant (if the first passes, the list is clean), but it earns its keep over time. Chunk failures have a habit of landing on ignore lists during incidents, "mute it until the CDN fix ships", and a suite-wide `WEBSHIP_JS_ERROR_IGNORE` entry would silence the general check here too. The pattern step reads the unfiltered list, so the sentinel keeps biting even after someone mutes the noise globally. And when it fails, the message says `JavaScript errors matched /chunk load error/`, triage starts with the diagnosis already written.

At scale: every project in a portfolio has one or two pages like this. Give the sentinel scenarios a shared tag (`@sentinel`) and run them first in CI, they are cheap, they fail fast, and a red `@sentinel` run tells the reviewer "known failure mode returned" before the full suite finishes. The tag also gives delivery managers a clean answer to "which known risks do we actively watch?": grep the portfolio for `@sentinel` and read the scenario names.

Recipe 442 Catching Synchronous Validation Errors

Problem

Your signup form rejects bad input with a clean red message, and also throws a synchronous error from the validation library on every rejection. Nothing visible breaks, so it shipped, and the team has been stepping around it for months. Then a real crash lands on the same page, and the engineer debugging it has to fish one genuine `TypeError` out of a console flooded with expected validation throws. Noise you tolerate today is camouflage for the bug you hunt tomorrow.

Solution

```
Scenario: Form rejects bad input without leaking errors
  Given I am on "/signup"
  When I fill in "email" with "not-an-email"
  And I press "Sign up"
  Then ".error-message" should be visible within 1 second
  And there should be no JavaScript errors
```

The timing works out because two waits stack in your favour. The visibility check is web-first, it polls every 100ms for up to the 1 second you asked for. And the press itself is a state-changing step, so webship-js's AfterStep hook has already run `smartSettle(page, 1500)` before the next step starts: ajax counter at zero, pending timers at zero, DOM quiet. By the time the final assertion reads the list, the whole validation cycle, synchronous throw included, has fired and been captured. A synchronous throw from the library arrives as a `pageerror` entry; a `console.error` it logs arrives at level `error`. Both fail the last step, which is exactly the contract: "show the user a friendly message AND stay silent in the console." Both halves matter.

Recipe 443 Differentiating Errors and Warnings

Problem

Your design system pins a framework version that prints deprecation warnings on every page, and the upstream fix is two quarters away. Meanwhile your own `TypeError`s must never ship. If the console policy cannot tell those two apart, it ends up in one of two bad places: a check that nags on every run until the team stops reading it, or no check at all. The Tuesday before release is exactly when someone proposes the second option.

Solution

```
  Scenario: Errors are forbidden, warnings ignored
    Given I am on "/"
    Then there should be no JavaScript errors
    # No warning check -- we accept deprecation warnings from third-party
libs
```

The scoping does its work at two layers. The explicit step only evaluates error entries, so warnings cannot fail it. And with the default `WEBSHIP_JS_ERROR_LEVELS=error`, warning-level console messages are never even captured, they cost no memory, no report space, and cannot leak into a failure message. The comment line is doing real work too: it tells the next reader the missing warning check is a decision, not a gap. When the upstream fix finally lands, tighten in two moves, add `there should be no JavaScript warnings` and widen capture with `WEBSHIP_JS_ERROR_LEVELS=error,warning` on the runs where you want it enforced.

Recipe 444 Watching for Specific Error Types

Problem

Your checkout throws a custom `BillingError` when the gateway declines a card, the error boundary catches it and shows the banner, exactly as designed. But the decline scenario *triggers* that error on purpose, so `there should be no JavaScript errors` can never pass here. The first person to hit this usually deletes the console check from the scenario entirely, and now the one page where money changes hands is the one page with no console guard. You want surgical tolerance: the deliberate `BillingError` is fine, anything else still fails.

Solution

```
  Scenario: Decline scenario emits exactly one BillingError
    Given the URL "**/api/payment" returns status 402 with body "Payment
Required"
    When I am on "/checkout"
    And I click "Place Order"
    Then ".error-banner" should be visible within 3 seconds
    And JavaScript errors should not match "^(?!.*BillingError).*"
```

Three details make this work. The mock comes first and uses a `page.route()` glob, `**/api/payment` matches the call wherever the page is served from, while a bare `/api/payment` would be compared against the full URL and never intercept anything. The 402 with a body gives the frontend a real decline to chew on. And the regex is the surgical part: `^(?!.*BillingError).*` anchors a negative lookahead at the start of each captured message, so the pattern matches any error that does **not** mention `BillingError` anywhere, and the step fails on a match. The deliberate error sails through; an accidental `TypeError` trips the wire. The step tests each captured message independently, so if the decline path throws `BillingError` *and* a stray `TypeError`, the `TypeError` still fails the scenario, which is precisely what you want.

Recipe 445 Asserting Warning Count

Problem

The dashboard carries three known deprecation warnings, documented, ticketed, waiting on a vendor release. The team's instinct is to ignore warnings wholesale until that lands, but "ignore warnings" is how three quietly becomes seventeen and nobody can say which fourteen are new. You want a budget: the known three pass, the fourth one fails the build and has to be explained in review.

Solution

Write a custom step over the world's captured list:

```
const { Then } = require('@cucumber/cucumber');
const { friendly } = require('./webship');

Then(/^there should be at most (\d+) JavaScript warnings?$/, function(max)
{
  const warnings = (this._jsErrors || []).filter((e) => e.type ===
'console.warning');
  if (warnings.length > parseInt(max, 10)) {
    throw friendly(`Expected at most ${max} warnings, got
${warnings.length}:\n${warnings.map(w => w.message).join('\n')}`);
  }
});
```

Then in the feature:

```
  Scenario: Dashboard warning budget is three
    Given I am on "/dashboard"
    Then there should be at most 3 JavaScript warnings
```

The world keeps one chronological list, `this._jsErrors`, with entries shaped `{ type, message }`, `pageerror` for uncaught exceptions, `console.warning` for warnings, and so on. The built-in `there should be no JavaScript warnings` step does exactly this filter with a threshold of zero; the custom step just makes the threshold a parameter. Two prerequisites: warnings must actually be captured (`WEBSHIP_JS_ERROR_LEVELS=error,warning`, the default `error` records none), and the step file must live in `tests/step-definitions/` so `require('./webship')` resolves and cucumber-js loads it.

At scale: budgets beat bans when you inherit projects in different states of repair. Onboard each project with its budget set to its current count, day-one green, no heroics required, then ratchet the number down as tickets close. A budget that never tightens is decoration; review the numbers quarterly the way you review dependency versions, and let the feature file diff (`at most 3` becoming `at most 1`) document the cleanup for free.

Recipe 446 Tracing Errors With Source Location

Problem

The captured list says `TypeError: t is not a function`. Minified, anonymous, owned by nobody, you cannot even file it against the right team yet. Before this becomes a debugging session, you want every scrap of context the capture pipeline holds: what kind of error it was, what it said in full, and what fired around it.

Solution

```
Scenario: Capture error source
  Given I am on "/"
  Then print JavaScript errors
```

Each printed entry is numbered in firing order and prefixed with its capture type: `[pageerror]` for uncaught exceptions, `[console.error]` for explicit logging, `[console.warning]` if you widened the levels. The prefix alone routes triage, an uncaught exception is a crash, while a

`console.error` is a developer's own message that survived to production; those go to different owners. The message text is everything the browser provided, untruncated, and the print step deliberately bypasses `WEBSHIP_JS_ERROR_IGNORE`, so you see muted noise too.

When the message itself is minified garbage, the fix lives in the app, not the test: serve source maps or an unminified build to the environment under test and the same step starts printing readable function names. To pin down *which step* introduced the error, pair this with the AfterStep counter from Recipe 448, the print tells you what fired, the counter tells you when.

Recipe 447 Asserting Only the First Page Error

Problem

One broken module fires three errors in a chain: the real failure, then two from code that expected the module to exist. A regression guard that matches against *all* captured errors is brittle here, refactor the page and the cascade reshuffles, and the test starts failing on a symptom instead of the cause. You want the assertion pinned to the error that starts the chain.

Solution

```
Scenario: First error matches expected pattern
  Given I am on "/broken"
  Then print JavaScript errors
```

In a custom assertion:

```
Then('the first JavaScript error should match {string}', function(pattern)
{
  const errors = this._jsErrors || [];
  if (!errors.length) throw friendly('No JavaScript errors recorded.');
  const re = new RegExp(pattern);
  if (!re.test(errors[0].message)) {
    throw friendly(`First error did not match /${pattern}/. Got:
${errors[0].message}`);
  }
});
```

The listeners push entries the instant events fire, so `this._jsErrors[0]` is chronologically first across both channels, uncaught exceptions and console output interleave in true order. Keep the `print` step in the scenario while the page is still under investigation; swap it for the custom assertion once you know which root-cause pattern to pin.

One subtlety worth knowing: the built-in assertion steps set a flag (`this._jsErrorsAsserted = true`) that tells the automatic end-of-scenario check "already handled, stand down." A custom step does not set it, so under `WEBSHIP_JS_ERROR_MODE=fail`, a scenario that deliberately triggers an error would pass your assertion and then fail the automatic check anyway. If the recorded errors are expected, set the flag inside your step.

Recipe 448 Hook-Level Console Recording

Problem

The print step tells you *what* fired; it doesn't tell you *when*. A twelve-step scenario with four captured errors leaves you bisecting by hand, commenting out steps, re-running, narrowing down. What you actually want is a log line that says "the count jumped right after the coupon click" so the guilty step names itself.

Solution

Use a step hook:

```
const { AfterStep } = require('@cucumber/cucumber');

AfterStep(function(step) {
  const errors = this._jsErrors || [];
  if (errors.length) {
    console.log(`After step "${step.pickleStep.text}", errors recorded:
${errors.length}`);
  }
});
```

This pinpoints which step introduced the first error, invaluable for cascading failures.

The timing is more reliable than it looks. Webship-js's own AfterStep hook runs `smartSettle(page, 1500)` after every state-changing step, waiting for the ajax counter, pending timers, and DOM mutations to go quiet, so the asynchronous fallout of a click (fetch callbacks, `setTimeout` chains) has usually landed in the list before your hook reads it. Errors get attributed to the step that caused them, not the step that happened to be running when a stray microtask flushed. Want it sharper? Stash the previous length on the world and log only the delta, a line that reads `+3 errors after "I click 'Apply Coupon'"` ends the bisection before it starts.

Recipe 449 Resetting Errors Mid-Scenario

Problem

A resilience scenario does something rude on purpose: it kills the payment API, watches the error banner appear, then retries and expects a clean recovery. The deliberate failure necessarily logs console errors, so a scenario-end console check can never pass, and the usual workaround (drop the check) throws away exactly the guarantee you care about: that the *recovery* path is silent. You need a line in the sand, not a blindfold.

Solution

```
When('I clear the recorded JavaScript errors', function() {
  this._jsErrors = [];
});
```

In the feature:

```
  Scenario: Recovery flow is clean after deliberate failure
    Given I am on "/checkout"
    When the URL "**/api/payment" returns status 500 with body "Internal
Server Error"
    And I click "Place Order"
    Then ".error-banner" should be visible within 3 seconds
    When I clear the recorded JavaScript errors
    And I click "Try again"
    Then ".success-banner" should be visible within 5 seconds
    And there should be no JavaScript errors
```

Errors and warnings share the one `this._jsErrors` list, so a single reset clears both. The clear step splits the scenario into two acts: everything before it may scream, everything after it must be silent. And because the automatic end-of-scenario check reads the same list, the reset also satisfies a suite running under `WEBSHIP_JS_ERROR_MODE=fail`, the policy only ever sees the post-reset entries. Order matters: clear *after* asserting the failure state (the banner check), never before, so the deliberate error is still on record while you verify the page handled it. The mock uses the `**/api/payment` glob so `page.route()` intercepts the call regardless of host, and the 500 carries a body so the frontend's error handler has something real to parse.

Recipe 450 The Standard Health Block

Pattern

Most teams settle on a four-line health block at the end of every scenario:

```
Then there should be no JavaScript errors
And there should be no broken images
And the page should pass an accessibility audit at level "AA"
And the page should have no critical accessibility violations
```

Together they cover: code correctness (no errors), asset correctness (no broken images), accessibility (level AA), and absence-of-blockers (no critical a11y violations).

Add the block to a `Background:` if you want it on every scenario in a file. Add it to a global `After` hook if you want it everywhere. The cost is small. The value is enormous on the day a deploy breaks something subtle.

The JavaScript-errors family and the friendly-error pipeline are the two features tester-friendliness depends on most: one says **what is wrong in the page**, the other says **what is wrong with the test in plain English**. Both reduce the gap between “my test failed” and “I know what to fix.”

Chapter 45: CI/CD Provider Deep-Dive

A test suite that doesn't run on every commit is a museum piece. The whole point of automated testing is to run automatically, in the background, on a machine that is always paying attention even when humans are not. Webship-js 2.0 ships ready-to-use CI configuration for **thirteen** providers, GitHub Actions, GitLab CI, Bitbucket Pipelines, Azure Pipelines, CircleCI, Travis, Jenkins, Drone, Semaphore, AWS CodeBuild, Google Cloud Build, Codefresh, and Bamboo. This chapter covers all thirteen, plus six more environments teams keep asking about (TeamCity, Woodpecker, Forgejo, Harness, Octopus, Codeship) where the same pattern ports in minutes.

This chapter is a recipe per provider. Each recipe shows: the moment you reach for it, the config file, the badge, the smallest possible setup, how the pieces fit, and the gotchas. Pick the one you actually use; skim the rest anyway, the day your company migrates providers, the skim pays for itself.

The Same Pattern, Everywhere

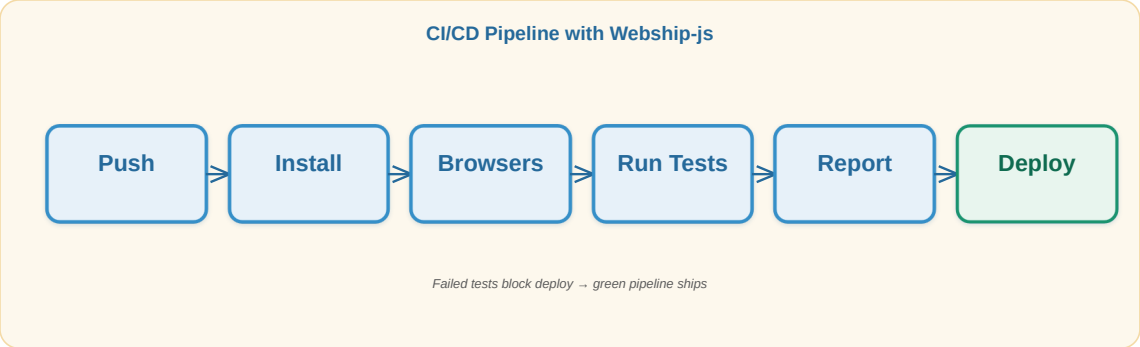


FIGURE 45.1 *The Same Pattern, Everywhere*

Every shipped config follows the same shape:

1. Install Node 20 (webship-js requires Node >= 20)
2. Install Playwright + Chromium with system deps
3. Background the app under test (the shipped examples run a fixture server on port 8080, the default `launchUrl` of `http://localhost:8080`)
4. Run `npm test`
5. Surface the cucumber-js report as a build artefact

That's it. No build step, no Docker daemon, no custom runners. `npm test` runs cucumber-js with the default profile: step definitions from `tests/step-definitions/**/*.js`, features from `tests/features/**/*.feature`, a 45-second step timeout, and a JSON report written to `tests/reports/cucumber_report.json`. The differences below are syntax, badge URLs, and per-provider quirks.

One thing to internalise before the recipes: everything that varies per environment is an environment variable, never a config edit. `LAUNCH_URL` points the suite at staging or production. `BROWSER` flips chromium/firefox/webkit. `WEBSHIP_VIDEO=on-failure` keeps failure videos.

`WEBSHIP_JS_ERROR_MODE=fail` turns console errors into build failures. The YAML stays identical across every project you own, only the variables change. That one design decision is what lets a single CI template serve forty repositories without forty forks of it.

Recipe 451 GitHub Actions

Problem: The suite passes on every laptop on the team, and then a new contributor's first PR breaks checkout in a way nobody notices for two days, because the tests only run when someone remembers to run them. You want every push and every pull request to trigger the suite on a machine nobody has to babysit, with the report waiting when a human finally looks.

File: `.github/workflows/github-actions.yml`. Free for public repos, 2,000 minutes/month for private repos. The single most popular CI/CD by a wide margin.

```
name: Webship-js Tests
on: [push, pull_request]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: '20'
      - run: npm install
      - run: npx playwright install --with-deps chromium
      - run: npm start &
      - run: sleep 3 && npm test
        env:
          FORCE_COLOR: 1
      - uses: actions/upload-artifact@v4
        if: always()
        with:
          name: cucumber-report
          path: tests/reports/
```

Badge:

```
[![Tests](https://github.com/USER/REPO/actions/workflows/github-actions.yml/badge.svg)](https://github.com/USER/REPO/actions)
```

How It Works: Each `- run:` line gets its own shell, but background processes survive between steps inside the same job, that is why `npm start &` in one step is still alive when `sleep 3 && npm test` runs in the next. `actions/setup-node@v4` pins Node 20 regardless of what the runner image happens to ship. `FORCE_COLOR: 1` makes cucumber-js emit ANSI colours into the log so failures are scannable. The artifact step uses `if: always()` because the report matters most when the job fails, artifacts are kept 90 days by default, tunable per step with `retention-days`.

Gotcha: Earlier `@v3` action versions emit Node-20 deprecation warnings. Bump to `@v4` to silence them. And remember `ubuntu-latest` is a moving target, it re-points to a newer OS image every couple of years, and the system libraries move with it. `--with-deps` is what carries Playwright through those migrations without you noticing.

At scale: Push every difference into `env:`, not into the YAML. A matrix over `BROWSER: [chromium, firefox, webkit]` triples coverage without a second workflow file, and `LAUNCH_URL` per environment lets the same workflow test staging on PRs and run production smoke checks after deploy. When you own dozens of repos, promote this file to an org-level reusable workflow (`workflow_call`), one fix to the template updates forty pipelines, instead of forty near-identical PRs.

Recipe 452 GitLab CI

Problem: Your organisation lives on GitLab, merge requests, issues, the container registry, all of it. The suite runs green on the lead's laptop and red on the shared runner, and now every MR review starts with the same tired question: "is that a real failure or the runner again?" You want the pipeline to be the referee, with the report attached as evidence on every merge request.

File: `.gitlab-ci.yml`. 400 compute-minutes/month on shared runners; unlimited on self-hosted. Strong for organisations already on GitLab.

```
image: node:20
stages: [test]
test:
  stage: test
  before_script:
    - npm install
    - npx playwright install --with-deps chromium
  script:
    - npm start &
    - sleep 3
    - npm test
  variables:
    FORCE_COLOR: "1"
  artifacts:
    when: always
    paths: [tests/reports/]
```

Badge:

```
[![pipeline](https://gitlab.com/USER/REPO/badges/main/pipeline.svg)]
(https://gitlab.com/USER/REPO/-/pipelines)
```

How It Works: `image: node:20` runs the entire job inside a Node container, so `before_script` installs dependencies and the Chromium binary into that container fresh on every run. The `script` block shares one shell, which is why a plain `npm start &` survives until `npm test` finishes. `artifacts: when: always` uploads `tests/reports/` on success and failure alike, and on gitlab.com, artifacts expire after 30 days by default, so set `expire_in` explicitly when your audit window is longer than your CI's memory.

Gotcha: Shared runners on GitLab.com are slower than GitHub's. For heavy suites, register a runner on your own hardware.

At scale: `node:20` is a floating tag, a routine image rebuild can move one project to a new Node minor overnight while its sibling projects stay behind, and suddenly "identical" suites disagree. Pin the image (`node:20.20` or a digest) and let `package-lock.json` pin Playwright, so every

project tests on the same browser build. Then move this whole job into a CI template repository and `include:` it from each project's `.gitlab-ci.yml`: the env vars carry the per-project differences, and a fleet-wide change is one commit in one repo.

Recipe 453 Bitbucket Pipelines

Problem: You are an Atlassian shop, Jira tickets, Bitbucket repos, Confluence specs. The QA lead wants test runs visible next to the pull request, and the free plan's 50 build-minutes are tight enough that every wasted rerun is a line item someone notices. You need the smallest pipeline that runs the suite, caches what it can, and keeps the report.

File: `bitbucket-pipelines.yml`. Free for small teams; 50 build-minutes/month on the free tier. Strong for Atlassian shops.

```
image: node:20
pipelines:
  default:
    - step:
        name: Test
        caches: [node]
        script:
          - npm install
          - npx playwright install --with-deps chromium
          - npm start &
          - sleep 3 && npm test
        artifacts: [tests/reports/**]
```

Badge:

```
[![Build](https://img.shields.io/bitbucket/pipelines/WORKSPACE/REPO/main)]
(https://bitbucket.org/WORKSPACE/REPO/pipelines)
```

How It Works: `caches: [node]` keys the dependency cache so `npm install` drops from minutes to seconds on warm runs, on a metered plan, the cache is not a nicety, it is the budget. The whole `script` runs in one container shell, so the backgrounded server survives until the tests

finish. `artifacts: [tests/reports/**]` keeps the report downloadable from the Pipelines UI; Bitbucket holds artifacts for 14 days, so anything you must keep longer belongs in external storage (the bucket pattern in Recipe 523 works here too).

Gotcha: Bitbucket's pipeline minutes count Linux 1:1 but Windows 4:1. Stick to Linux for cost.

At scale: Do the minutes math before you commit the fleet: a 5-minute suite running nightly is roughly 150 minutes a month *per project*, the free tier covers one project running weekly, not ten running nightly. Teams that stay on Bitbucket buy minutes or attach a self-hosted runner; teams that don't usually keep the repos and move CI elsewhere. Either way the webship-js side never changes, only the YAML wrapper does, which is exactly the point of this chapter.

Recipe 454 CircleCI

Problem: Your suite takes long enough on slow shared runners that the team has stopped waiting for it, they merge first and check the result later, which quietly defeats the entire point of CI. CircleCI's fast container start and generous free tier are the draw; a config that doesn't fight its execution model is the price of admission.

File: `.circleci/config.yml`. 6,000 build-minutes/month free. The shipped config uses `cimg/node:20.20`, the lightweight image dropped `apt upgrade` and unused tools for faster start.

```
version: 2.1
jobs:
  test:
    docker:
      - image: cimg/node:20.20
    steps:
      - checkout
      - run: npm install
      - run: npx playwright install --with-deps chromium
      - run:
          command: npm start
          background: true
      - run: sleep 3 && npm test
      - store_artifacts:
          path: tests/reports
workflows:
  build:
    jobs: [test]
```

Badge:

```
[![CircleCI](https://circleci.com/gh/USER/REPO.svg?style=shield)]
(https://circleci.com/gh/USER/REPO)
```

How It Works: CircleCI runs every `run` step in its own shell and does not let stray processes drift across steps, that is why the fixture server uses CircleCI's native `background: true` instead of a bare `&`, which would die with its step's shell. The `cimg/node:20.20` image is pinned to an exact Node minor, which is the quiet hero of this config: the runtime cannot drift under you between Tuesday and Thursday. `store_artifacts` exposes `tests/reports` in the job UI, retained for 30 days.

Gotcha: The `cimg/node` image includes browsers system deps, no `--with-deps` needed. The shipped config still passes the flag for safety on custom images.

At scale: That pinned image tag is the policy worth copying to every provider you run: pin the image, pin Playwright (1.58 in webship-js 2.0), and upgrade both deliberately, on your schedule, across all projects in one coordinated wave. Unpinned images are how "nothing changed but three suites started flaking" arrives at 2 a.m., a fresh browser build alters a focus or scroll behaviour, and the diff you are hunting for does not exist in any of your repos.

Recipe 455 Travis CI

Problem: You inherit a repository with a `.travis.yml` from 2019 and a README badge stuck on "build: unknown". The team asks whether to revive it or migrate, and nobody wants to spend a sprint finding out. This recipe is the honest answer: here is the working config, and here is why you probably shouldn't invest in it.

File: `.travis.yml`. The shipped config replaces the long-defunct `.travis-ci.yml` (`org` is dead; `com` is the survivor).

```
language: node_js
node_js: [20]
addons:
  chrome: stable
script:
  - npm install
  - npx playwright install --with-deps chromium
  - npm start &
  - sleep 3 && npm test
```

Badge:

```
[![Build Status](https://app.travis-ci.com/USER/REPO.svg?branch=main)]
(https://app.travis-ci.com/USER/REPO)
```

How It Works: `language: node_js` provisions the Node 20 toolchain before your scripts run. The `addons: chrome: stable` line predates Playwright bundling its own browsers and is harmless but redundant, `npx playwright install --with-deps chromium` is what actually

provides the browser the tests drive. All `script` lines share one shell, so the standard background-server pattern works unchanged.

Gotcha: Travis's free tier was discontinued for open source in 2021. Most projects migrate off. The shipped config exists for legacy reasons; new projects should pick GitHub Actions. If you maintain a portfolio, treat every `.travis.yml` you find as a migration ticket waiting to be filed, not a system to extend.

Recipe 456 Jenkins

Problem: Your company runs Jenkins on its own iron, compliance wants builds inside the network, and nobody is migrating two hundred jobs to a SaaS this quarter. You get unlimited minutes and total control; in exchange you own Node installs, agent hygiene, and every plugin upgrade. You want the smallest Jenkinsfile that runs the suite and archives the evidence.

File: `Jenkinsfile`. Self-hosted, no usage limits, but you own the runners.

```
pipeline {
  agent any
  tools { nodejs '20' }
  stages {
    stage('Install') { steps { sh 'npm install' } }
    stage('Browsers') { steps { sh 'npx playwright install --with-deps
chromium' } }
    stage('Test') {
      steps {
        sh 'npm start &'
        sh 'sleep 3 && npm test'
      }
    }
  }
  post {
    always {
      archiveArtifacts artifacts: 'tests/reports/**', allowEmptyArchive:
true
    }
  }
}
```

Badge: Jenkins doesn't expose a public badge by default. Install the Jenkins Build Status plugin and point a shields.io endpoint at it.

How It Works: Each `sh` step runs in its own shell on the agent. The declarative `post { always { archiveArtifacts ... } }` block is Jenkins' answer to `if: always()`, the report is attached to the build page whether the suite passed or not, and retention follows the job's build-discarder policy, which (unlike SaaS providers) you control completely. `tools { nodejs '20' }` resolves Node from a Global Tool Configuration entry at runtime, so agents don't need Node baked into their images.

Gotcha: Jenkins agents need Node 20 installed (via NodeJS plugin). The `tools { nodejs '20' }` block references a Global Tool configuration, set it up in `Manage Jenkins → Global Tool Configuration`. One more Jenkins-specific trap: a bare `sh 'npm start &'` can leave the step waiting on the open log pipe, or get the server reaped by Jenkins' process cleanup. If the Test stage hangs or hits connection-refused, launch the server with `nohup`, redirect its output, and set `JENKINS_NODE_COOKIE=dontKillMe` on it.

At scale: Forty projects means forty Jenkinsfiles drifting apart unless you stop it early. Put this pipeline in a Jenkins Shared Library, reduce each repo to a one-line `webshipPipeline()` call, and pass the per-project differences, `LAUNCH_URL`, the tag expression to run, as parameters. Agent hygiene is the other half: label the agents that have browser system libraries and pin webship jobs to that label, or `--with-deps` will fail on the one agent in the pool that's missing a package, and it will always be the 3 a.m. run that lands there.

Recipe 457 Azure Pipelines

Problem: Your .NET services already build in Azure DevOps, boards, repos, releases, the lot, and the release manager keeps asking why the UI tests live somewhere else. Every release review means stitching evidence from two systems, and the one time it mattered, the link to the external test run had expired. You want the webship-js suite in the same pipeline view as the deployment it guards.

File: `azure-pipelines.yml`. 1,800 free minutes/month for public projects; private projects need a paid plan after the free hosted parallel job expires.

```
trigger: [main]
pool:
  vmImage: 'ubuntu-latest'
steps:
- task: NodeTool@0
  inputs: { versionSpec: '20.x' }
- script: npm install
- script: npx playwright install --with-deps chromium
- script: npm start &
- script: sleep 3 && npm test
  env: { FORCE_COLOR: 1 }
- task: PublishPipelineArtifact@1
  condition: always()
  inputs:
    targetPath: 'tests/reports'
    artifactName: 'cucumber-report'
```

Badge:

```
[![Build](https://dev.azure.com/ORG/PROJECT/_apis/build/status/REPO)]
(https://dev.azure.com/ORG/PROJECT/_build)
```

How It Works: `NodeTool@0` resolves Node 20.x at runtime, so the suite is not hostage to whatever the `ubuntu-latest` image happens to carry this quarter. Each `- script:` step gets its own shell, but they all run on the same hosted agent, so the backgrounded fixture server from one step is still listening when the next step runs the tests, the same model as GitHub Actions. `PublishPipelineArtifact@1` with `condition: always()` is the piece teams forget: without it, a red run uploads nothing, and a red run is precisely when you need `tests/reports/` the most.

Gotcha: First-time setup requires creating an Azure DevOps organisation. The free parallel job request takes 2-3 business days to be approved by Microsoft.

At scale: the unit Microsoft meters is the *parallel job*, granted per organisation, not per project. One free hosted job means your forty pipelines run one at a time, a nightly fleet that should finish by 2 a.m. is still queuing at breakfast. Either buy parallel jobs or register self-hosted agents (and if you self-host, bake the Playwright system deps into the agent image so `--with-deps` stops being a per-run download). Put `LAUNCH_URL` and friends in a variable group in the Azure DevOps Library: one shared set of environment wiring, linked into every pipeline, instead of forty copies drifting apart.

Recipe 458 AWS CodeBuild

Problem: Security finally signs off on automated testing, with one condition: the staging environment stays inside the VPC, unreachable from any SaaS runner on the public internet. Your infrastructure is already Terraform-on-AWS, the team already lives in CloudWatch, and standing up a separate CI vendor for one test suite is a procurement conversation nobody wants. CodeBuild runs inside your account, inside your network, on your IAM roles.

File: `buildspec.yml`. Pay-per-minute. The lower compute classes are the cheapest; `BUILD_GENERAL1_SMALL` is plenty for webship-js.

```
version: 0.2
phases:
  install:
    runtime-versions: { nodejs: 20 }
    commands:
      - npm install
      - npx playwright install --with-deps chromium
  build:
    commands:
      - npm start &
      - sleep 3 && npm test
artifacts:
  files: [tests/reports/**/*]
```

Badge: AWS CodeBuild does not ship native badges. Wire a CloudWatch alarm to an SNS topic and surface via a shields.io endpoint or a custom dashboard.

How It Works: `runtime-versions: nodejs: 20` in the install phase selects the managed Node runtime, no setup task, no curl-pipe-bash. Each command in a buildspec gets a fresh shell, but every command shares the same build container, so the backgrounded `npm start` keeps serving while `sleep 3 && npm test` runs as the next command. The `artifacts` block hands `tests/reports/**/*` to the S3 bucket configured on the CodeBuild project, which means report retention is not a CI setting here at all, it is whatever lifecycle policy you put on that bucket. For VPC-internal staging, attach the project to the VPC and point `LAUNCH_URL` at the internal hostname; the suite neither knows nor cares that no public route exists.

Gotcha: CodeBuild has no built-in scheduler. Trigger from EventBridge (cron) or from a webhook on your source provider (CodeCommit, GitHub, Bitbucket).

At scale: pay-per-minute makes the fleet cost an arithmetic problem instead of a plan-tier surprise: forty suites at five minutes nightly on `BUILD_GENERAL1_SMALL` is a predictable, boring number, and boring is what you want when finance asks. Keep it boring by splitting runs the way the tag chapter teaches: `@critical` on every push, the full suite on the EventBridge nightly. And because artifacts land in S3 anyway, this is the one provider where long-term evidence storage needs no extra step, one lifecycle rule (transition to infrequent access at 90 days) covers every project in the account.

Recipe 459 Google Cloud Build

Problem: The application deploys to Cloud Run, secrets live in Secret Manager, and the team authenticates with workload identity instead of exported keys, then the test suite shows up needing a service-account JSON file pasted into a third-party CI vendor, and the platform engineer vetoes it on the spot. Keeping the suite inside the same GCP project ends the credential-export argument before it starts.

File: `cloudbuild.yaml`. 120 free build-minutes per day. Pay-per-minute beyond that. Tight integration with the rest of Google Cloud.

```
steps:
- name: 'node:20'
  entrypoint: 'bash'
  args:
    - '-c'
    - |
      npm install
      npx playwright install --with-deps chromium
      npm start &
      sleep 3
      npm test
artifacts:
  objects:
    location: 'gs://YOUR_BUCKET/cucumber-reports/'
    paths: ['tests/reports/**']
```

Badge: Not native. Use a shields.io endpoint that queries Cloud Build's REST API.

How It Works: The whole job is one step running the `node:20` container with a `bash` entrypoint, and the `|` block hands bash a single multi-line script, one shell, so `npm start &` survives until `npm test` exits, no provider-specific background mechanism needed. The `artifacts.objects` block copies `tests/reports/**` straight into a GCS bucket, which (like CodeBuild's S3 pattern) moves retention out of the CI tool and into a bucket lifecycle rule you control. Pin the image to a digest rather than the floating `node:20` tag when you need build-to-build reproducibility.

Gotcha: Cloud Build's default machine type is small. For Playwright-heavy suites, set `options.machineType: 'E2_HIGHCPU_8'`.

At scale: the 120 free daily minutes are per billing project, and a ten-suite nightly fleet eats them before midnight, after that you are paying per minute, so the machine-type gotcha becomes a cost dial: `E2_HIGHCPU_8` costs more per minute but finishes Playwright work so much faster that the bill often comes out flat. Use substitution variables (`_LAUNCH_URL`, `_BROWSER`) in one shared

`cloudbuild.yaml` and set them per trigger; every product gets its own staging URL and browser without forking the file, which is the same env-var-not-config-edit rule this chapter opened with.

Recipe 460 TeamCity

Problem: The company has run TeamCity on-prem for a decade, every Java service builds there, the release dashboard lives there, and ops knows its failure modes by heart. Then the web team adopts webship-js, and suddenly the QA suite is the one build that lives somewhere else: invisible on the dashboard, absent from the release gate. Adding one more vendor for one suite is a harder sell than teaching TeamCity to run Node.

File: `.teamcity/settings.kts` (plus `pom.xml`). Kotlin DSL. JetBrains' on-prem CI, common in JVM shops.

```
import jetbrains.buildServer.configs.kotlin.v2019_2.*

version = "2024.03"
project {
  buildType(BuildType {
    id("Webship_Tests")
    name = "Webship-js Tests"
    steps {
      script {
        name = "Install + Test"
        scriptContent = """
          npm install
          npx playwright install --with-deps chromium
          npm start &
          sleep 3
          npm test
        """.trimIndent()
      }
    }
    artifactRules = "tests/reports => cucumber-report"
  })
}
```

Badge: TeamCity exposes a status URL but no SVG. Wrap via shields.io endpoint.

How It Works: The Kotlin DSL is configuration-as-code in the literal sense, the build definition lives in the repo, so a change to how tests run is a reviewable diff instead of a click someone made in the admin UI last spring. `scriptContent` is one script in one shell, which is why the plain `npm start &` background pattern works unmodified. `artifactRules = "tests/reports => cucumber-report"` publishes the report directory under a stable artifact name; retention follows the server's clean-up rules, which on-prem means *your* rules. The build agent must have Node 20 available, TeamCity will not provision it for you the way `setup-node` does.

Gotcha: The Kotlin DSL needs the TeamCity server to be on a recent version (2019.2+). Older on-prem instances may need the XML format instead.

Recipe 461 Semaphore

Problem: Four in the afternoon, six PRs queued, and the team has developed the habit that quietly kills CI cultures: merge now, check the pipeline later, because waiting for it costs more attention than ignoring it. The suite is not slow, the feedback loop is. Semaphore's pitch is raw start-up speed: machines boot in seconds, so the time between push and verdict shrinks to something a developer will actually sit through.

File: `.semaphore/semaphore.yml`. 1,300 free minutes/month. Fast Docker-based provider with a clean UI.

```
version: v1.0
name: Webship-js
agent:
  machine:
    type: e1-standard-2
    os_image: ubuntu2204
blocks:
- name: Test
  task:
    jobs:
    - name: cucumber
      commands:
      - checkout
      - npm install
      - npx playwright install --with-deps chromium
      - nohup npm start &
      - sleep 3
      - npm test
    epilogue:
      always:
        commands:
          - artifact push job tests/reports
```

Badge:

```
[![Build](https://USER.semaphoreci.com/badges/REPO/branches/main.svg)]
(https://USER.semaphoreci.com/projects/REPO)
```

How It Works: The `agent` block picks the VM size (`e1-standard-2`) and OS image (`ubuntu2204`) explicitly, there is no `ubuntu-latest` ambiguity to drift under you. `checkout` is Semaphore's built-in clone command, the job's commands share a session, and the `epilogue.always` block plays the same role as `if: always()` elsewhere: `artifact push job tests/reports` uploads the report whether the suite passed or failed. Job artifacts are the short-lived tier; promote anything you must keep to project-level artifacts or external storage.

Gotcha: Semaphore uses a long-lived `nohup` for background processes, `npm start &` alone can be reaped.

Recipe 462 Drone CI

Problem: The compliance ruling is blunt: source code and build logs never leave the network. Jenkins would do it, but nobody on the team wants to inherit a plugin garden for one Node test suite, you want something you can run as a single container, configure with one YAML file, and explain to the auditor in a sentence. That is Drone's whole personality.

File: `.drone.yml`. Open-source, self-hosted. Increasingly used as a lightweight alternative to GitLab CI.

```
kind: pipeline
type: docker
name: webship-js
steps:
- name: test
  image: node:20
  commands:
  - npm install
  - npx playwright install --with-deps chromium
  - npm start &
  - sleep 3 && npm test
```

Badge: Use a shields.io endpoint pointed at the Drone server's status API.

How It Works: `type: docker` means every step is a container; the `test` step pulls `node:20`, runs all four commands in one shell inside it, and throws the container away. That is the cleanest isolation model in this chapter, nothing persists between runs except what you cache deliberately, and it is also why the background-server pattern works: server and tests share one container, one shell, one lifetime.

Gotcha: Drone has been quiet on releases since the Harness acquisition. Many users have moved to Woodpecker (a Drone fork; see next recipe). The same `.drone.yml` works in both.

At scale: self-hosting puts you on the other side of the image-supply problem. Forty repos each pulling `node:20` from Docker Hub on every run will meet the rate limiter at the worst possible hour, mirror the image in an internal registry and pin it by digest, so every project tests on the byte-identical runtime and the 2 a.m. run cannot be broken by an upstream rebuild. The portability is the other payoff: the same `.drone.yml` runs under Woodpecker verbatim, so the migration path off a stagnating server is a DNS change, not a rewrite.

Recipe 463 Woodpecker CI

Problem: The annual security review flags your Drone server: release cadence near zero since the Harness acquisition, CVE response unclear, and the auditor wants a maintenance story you cannot give. Migrating CI usually means a quarter of rewrites, except here, where the community fork was built to accept your existing config unchanged.

File: reuses `.drone.yml`. A community fork of Drone, MIT-licensed and actively maintained.

```
# Same as Drone's .drone.yml -- Woodpecker accepts it verbatim
when:
  branch: [main]
steps:
  test:
    image: node:20
    commands:
      - npm install
      - npx playwright install --with-deps chromium
      - npm start &
      - sleep 3 && npm test
```

Badge:

```
[![status](https://WOODPECKER_SERVER/api/badges/USER/REPO/status.svg)]
(https://WOODPECKER_SERVER)
```

How It Works: Same execution model as Drone, each step is a disposable container, commands share one shell inside it, with a `when: branch` filter shown here so the suite runs on `main` pushes rather than every branch. The webship-js side is untouched by the fork: `LAUNCH_URL`, `BROWSER`, and the rest of the env-var surface behave identically, because the suite never knew which server was running it in the first place.

Gotcha: Drone's "kind/type" block is optional in Woodpecker but still parses correctly. New Woodpecker projects can use the leaner native syntax.

Recipe 464 Forgejo Actions

Problem: The organisation moved its repos to a self-hosted Forgejo instance, data sovereignty, no US cloud dependency, the usual reasons, and the migration script carried everything across except one thing that matters daily: the `.github/workflows/` directory that used to run the tests. Rewriting forty working workflows for a new CI dialect is exactly the tax the migration plan promised to avoid.

File: reuses `.github/workflows/*`. Forgejo (the Codeberg-driven Gitea fork) has a GitHub-Actions-compatible runner.

```
# Identical syntax to GitHub Actions; runs unmodified.
name: Webship-js Tests
on: [push, pull_request]
jobs:
  test:
    runs-on: docker
    container:
      image: node:20
    steps:
      - uses: https://code.forgejo.org/actions/checkout@v4
      - run: npm install && npx playwright install --with-deps chromium
      - run: npm start &
      - run: sleep 3 && npm test
```

Badge:

```
[![status]
(https://FORGEJO_HOST/USER/REPO/actions/workflows/main.yml/badge.svg)]
(https://FORGEJO_HOST/USER/REPO/actions)
```

How It Works: The Forgejo runner speaks the GitHub Actions protocol, so the workflow syntax, jobs, steps, `uses:` , `run:` , carries over unmodified. Two deliberate differences in the file above: `runs-on: docker` with an explicit `container.image: node:20` replaces GitHub's hosted `ubuntu-latest` (you own the runner, so you own the image), and the checkout action is pinned to its full `code.forgejo.org` URL so the runner never silently reaches out to a registry you did not choose. Everything below the YAML, the install, the backgrounded server, `npm test` , the report, is byte-for-byte the GitHub Actions recipe.

Gotcha: Forgejo's action runner uses the same protocol as GitHub but pulls action images from `code.forgejo.org` by default. Pin actions explicitly if you mix runners.

Recipe 465 Harness CI

Problem: Leadership consolidates: one platform for CI, CD, and feature flags, signed at the executive level, and every team's pipelines are expected to move in by Q3. You did not pick Harness, it picked you. The question is not "is this the right CI" but "what is the smallest pipeline definition that gets the suite running inside it without disturbing the suite itself".

File: `.harness/webship-js-pipeline.yml`. Enterprise CI/CD with strong feature-flag and deployment tooling baked in.

```
pipeline:
  name: Webship-js Tests
  identifier: webship_js
  stages:
  - stage:
      name: Test
      type: CI
      spec:
        execution:
          steps:
          - step:
              type: Run
              name: Test
              spec:
                connectorRef: docker-hub
                image: node:20
                command: |-
                  npm install
                  npx playwright install --with-deps chromium
                  npm start &
                  sleep 3 && npm test
```

Badge: Harness offers status badges via its API; configure a public dashboard or embed via shields.io.

How It Works: The nesting is deep, pipeline, stage, execution, step, but the working core is one `Run` step: pull `node:20` through the `connectorRef` (Harness's named credential for Docker Hub), run the familiar four commands in one shell. The `command: |-` block is the same single-shell pattern as Cloud Build, so the backgrounded server needs no special handling. The execution itself happens on a *delegate*, an agent you run in your own cluster or VM, which is why the suite can reach internal staging URLs the SaaS control plane never sees.

Gotcha: Harness pipelines bind tightly to its delegate model, you'll need a delegate running in the cluster or VM that executes the build.

Recipe 466 Bamboo Data Center

Problem: The Bamboo Data Center renewal just went through for three more years, the build agents sit in the company server room, and the change-advisory board is not entertaining a CI migration this cycle. The web team still needs its webship-js suite running per push. The pragmatic move is to meet the estate where it is: a Bamboo Specs file in the repo, the same suite underneath.

File: `bamboo-specs/bamboo.yml`. Atlassian's on-prem CI for enterprise shops still on the Atlassian stack.

```
version: 2
plan:
  project-key: WS
  key: TEST
  name: Webship-js Tests
stages:
- Test:
  - jobs:
    - Test:
        tasks:
        - script:
            scripts:
            - |
              npm install
              npx playwright install --with-deps chromium
              npm start &
              sleep 3 && npm test
        artifacts:
          - name: cucumber-report
            location: tests/reports
            pattern: '**'
```

Badge: Bamboo exposes a JSON status endpoint, wire to shields.io.

How It Works: Bamboo Specs is Bamboo's configuration-as-code: the YAML maps to a project (`WS`), a plan (`TEST`), and a script task whose multi-line block runs in one shell, background-server pattern intact. The `artifacts` entry publishes `tests/reports` with pattern `'**'` so the whole report directory (HTML, JSON, screenshots) attaches to each build result; retention fol-

lows the plan's expiry settings, which on Data Center are yours to set as generously as the audit window demands. As with TeamCity and Jenkins, the agent must already have Node 20, provision it in the agent image, not in the plan.

Gotcha: Atlassian deprecated Bamboo Cloud years ago. Only the Data Center / Server editions remain. Any new project should consider Bitbucket Pipelines instead, which is the modern Atlassian CI offering.

Recipe 467 Codefresh

Problem: Everything else in the platform is GitOps already, deployments reconcile through Argo, environments are declared, drift is a dirty word, and the test suite is the last piece still running on a VM-shaped CI that thinks in shell scripts. The platform team wants CI that speaks containers natively, so the test pipeline composes like the rest of the stack.

File: `codefresh.yml`. Container-native CI/CD with first-class GitOps support.

```
version: '1.0'
stages: [prepare, test]
steps:
  install:
    title: Install dependencies
    image: node:20
    stage: prepare
    commands:
      - npm install
      - npx playwright install --with-deps chromium
  test:
    title: Run tests
    image: node:20
    stage: test
    commands:
      - npm start &
      - sleep 3 && npm test
```

Badge: Native badge support via project settings.

How It Works: Every step is a container, but, unlike Drone, where each step's container is disposable and isolated, Codefresh mounts one shared volume across all steps in the pipeline. That is the trick that makes this two-step layout work: the `install` step's `node_modules` and browser download are still on disk when the `test` step starts in a fresh `node:20` container. Within the `test` step, both commands share a shell, so the backgrounded fixture server lives exactly as long as the tests need it.

Gotcha: Codefresh's free tier removed unlimited builds in 2023, check current pricing for non-OSS use.

Recipe 468 Octopus Deploy (CD-Only)

Problem: The release manager runs deployments through Octopus and asks the reasonable-sounding question: "can Octopus run the tests too, so everything lives in one tool?" Someone has to give the honest answer before a sprint disappears into making a deployment tool impersonate a CI server.

File: no test runner, Octopus is deployment-only. Webship-js tests are invoked from upstream CI (GitHub Actions, Jenkins, etc.) and Octopus deploys the validated artefact.

Pattern:

```
GitHub Actions  →  npm test  →  upload artefact  →  Octopus Deploy
                                                    (downloads artefact +
ships)
```

Why list it here: Many teams ask "can Octopus run my tests?". Strictly no, Octopus runs deployments, not test suites. Pair it with a CI provider that runs webship-js, then deploy the green build through Octopus. The division of labour is actually a feature: CI produces the green run *and* the `tests/reports/` evidence, the artefact Octopus deploys is the one that evidence describes, and the release page can link straight to the report that justified shipping it. A post-deploy smoke run, the same suite with `LAUNCH_URL` pointed at production and a `@smoke` tag filter, still belongs to the CI side, triggered by Octopus's deploy hook.

Recipe 469 Codeship (Retired)

Problem: A repo you inherit still carries `codeship-steps.yml`, a badge that resolves to nothing, and a README that swears the tests run on every push. They have not run since 2023.

File: none, Codeship was retired in 2023.

Action: If you have a legacy `codeship-steps.yml` or `codeship-services.yml`, migrate to CloudBees CI (the post-acquisition successor) or pick any of the providers above. The translation is mechanical, a Codeship config maps straight onto the five-step shape from the top of this chapter, and deleting the dead config matters as much as adding the new one, because a badge that lies is worse than no badge at all.

Recipe 470 Choosing a Provider

The question rarely arrives as “which CI is best”. It arrives as a merger, a compliance ruling, a pricing change, or a new platform lead with opinions, and suddenly someone has to pick, this week, for every project at once. The good news: because the suite’s entire per-environment surface is environment variables, the choice is lower-stakes than it feels. You are picking a YAML wrapper, not re-architecting your tests.

The decision tree, simplified:

```
Public repo?                → GitHub Actions (free, unlimited)
GitLab shop?                → GitLab CI (free shared runners, unlimited
self-hosted)
Atlassian shop?             → Bitbucket Pipelines (smaller suites) or
Bamboo Data Center
Self-host required?         → Jenkins, Drone, Woodpecker, or Forgejo
On Azure?                   → Azure Pipelines
On AWS?                     → CodeBuild (paired with CodePipeline)
On GCP?                     → Cloud Build
JVM shop, on-prem JetBrains? → TeamCity
Container-native + GitOps? → Codefresh or Harness
Just want fast results?     → Semaphore or CircleCI
```

The shipped configs let you pivot. Drop a new file, push, done. No re-implementation needed.

At scale: pick *one* provider for the portfolio and defend that choice harder than the choice itself. Twenty projects on one provider means one template, one secrets convention, one place to pin browser images, and one bill to explain; twenty projects on four providers means four of everything and a platform engineer who spends Fridays translating YAML dialects. When an acquisition hands you repos on a second provider, run both only as long as the migration takes, and let the env-var rule do the moving: if a suite needs anything beyond `LAUNCH_URL`, `BROWSER`, and friends to run somewhere new, that is a defect in the suite, not in the provider.

CI/CD Gotchas Common to All Providers

1. Background server reaping: Some providers kill background processes between steps. Use `nohup`, `&`, or `disown` per the provider's docs.

2. Sleep before test: Always give the fixture server time to bind: `sleep 3 && npm test`. Without it, the first test hits a refused connection and fails for the wrong reason.

3. Browser system deps: `npx playwright install --with-deps chromium` installs both the browser binary and the system libraries it needs. Skip `--with-deps` only on images you know already have them (cimg/node, mcr.microsoft.com/playwright).

4. Color output: Set `FORCE_COLOR=1` in the job environment so cucumber-js emits ANSI colours. Without it, the report is hard to read.

5. Artefacts always: Upload reports on success and failure, otherwise you lose the failure evidence when you need it most. Most provider configs accept `always()` or `if: always:` for this.

6. Cache `node_modules`: Most providers key the dependency cache on a hash of `package-lock.json`. Adds ~30s the first run; saves a minute every run after, multiplied by every run of every project, which is where metered plans live or die.

The shipped configs already handle all six. If you adapt them for a provider not in this chapter, port the same six concerns, you'll skip an hour of debugging.

A Note on Reports

Every provider config above uploads `tests/reports/` as a build artefact. Inside that directory you'll find:

- `cucumber_report.html`, human-readable report with embedded screenshots
- `cucumber_report.pdf`, print-friendly version
- `cucumber_report.json`, machine-readable (parse for dashboards)
- `screenshots/`, failure screenshots
- `videos/`, (if recording enabled)

Point your post-CI tools at the JSON file. Push the HTML to S3, GCS, or a static-hosting service for browser-friendly access. The report is the most valuable artefact your suite produces, preserve it.

One more habit that pays off across a portfolio: stamp the report with run metadata. `bin/generate-reports.js` takes `--app-version`, `--env`, `--browser`, `--platform`, and arbitrary `--metadata Key=Value` pairs, wire your CI variables into those flags and every report says exactly which build, environment, and browser produced it. That is what turns forty JSON files into one comparable dashboard, and it is the difference between “the suite failed” and “release 2.4.1 failed on staging, Firefox only” when a manager asks.

CI/CD is the bridge between writing tests and trusting them. With thirteen ready-to-use configs shipped in the repo and six more environments mapped in this chapter, the bridge is already built. Pick the one that matches the runners you have, push, and watch the green.

Chapter 46: Video Recording Deep-Dive

A failing test that prints a stack trace is useful. A failing test that prints a stack trace **and** a 30-second video of what the browser was doing is invaluable. A flake that happens only on the CI runner, not on your laptop, not on the dev VM, becomes solvable the moment you can watch it. Webship-js 2.0 added a native `recordVideo` integration: a `WEBSHIP_VIDEO` environment variable with four modes (`off`, `on`, `on-failure`, `tag`), `@video` and `@no-video` tags for per-scenario overrides, and start/stop/save steps wired to Playwright's `BrowserContext.recordVideo` options.

This chapter is the full deep-dive: configuration, retention policy, sizes, hook patterns, CI-artefact wiring, and the trade-offs between recording everything and recording only failures.

How Playwright Records Video

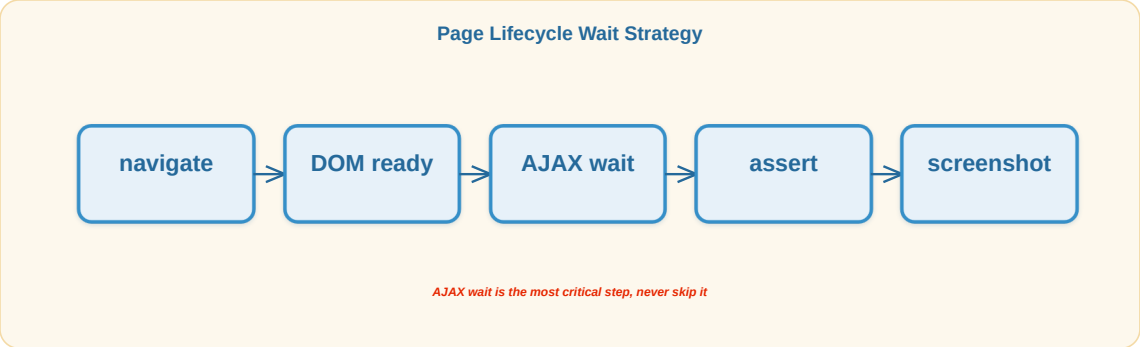


FIGURE 46.1 *How Playwright Records Video*

Playwright's video recorder runs at the browser context level, not the page level. When `recordVideo.dir` is set on the context options, every page in that context streams to a `.webm` file. The recording starts when the page is created and stops when the context closes, then, and only then, does Playwright flush the file to disk. There is no "pause", no "resume", and no retroactive capture. Recording is a property of the context, decided at the moment the context is built. That one fact explains almost everything else in this chapter, including the warning that closes this section.

Webship-js wraps this with an environment-variable control surface and a handful of Gherkin-friendly steps, so you rarely touch the raw Playwright options. The env vars first:

- `WEBSHIP_VIDEO`, `off` (the default), `on` (record every scenario), `on-failure` (record everything, then keep only the runs that failed), or `tag` (record only `@video`-tagged scenarios). One variable, four shapes, set per environment instead of edited per project.
- `WEBSHIP_VIDEO_DIR`, where files land; defaults to `./videos`. The directory is created if it does not exist.
- Recording size defaults to 1280x720. Files are named by the pattern `{datetime}.{feature_file}.{scenario}.{status}.{ext}`, timestamp, source feature, scenario name, and pass/fail status, all readable from a directory listing without opening a single file. That status token in the filename is the difference between scanning a folder and watching twenty videos to find the one failure.

Two tags override the global mode per scenario, no code required: `@video` forces recording on even when the mode is `off`, and `@no-video` suppresses it even when the mode is `on`. And the steps you will use most:

- `Given I start video recording`, creates a recording-enabled context if one isn't already active
- `When I stop video recording`, finalises the current recording and flushes the `.webm` to disk
- `When I save the current video as "name"`, moves the finished recording to a chosen filename

A fresh context is created per scenario by default, so videos are naturally per-scenario, one scenario, one file, no bleed between tests.

Now the warning, because it bites everyone exactly once and then they remember it forever. Since recording is fixed at context creation, the mid-scenario `start` / `stop` steps cannot flip a switch on a live context. They tear the current context down and build a new one. **Page state is lost the moment you call them**: your logged-in session, the cookies in `context.cookies()`, localStorage, the current URL, the DOM you just settled. If you `start video recording` after logging in, the new context is logged out and sitting on a blank page. The rule that follows from this is simple, make `Given I start video recording` the *first* step of the scenario, before any navi-

gation or authentication, or set `WEBSHIP_VIDEO=on` and let the per-scenario context record from birth so you never call the mid-scenario steps at all. Everything that happened before the start step is gone, and no error will tell you so; you will just see a video that begins from a clean slate.

Recipe 511 Recording an Entire Scenario

Problem

Thursday's sprint review needs a walkthrough of the booking flow, and you are about to screen-record yourself clicking through it by hand, for the fourth sprint in a row, because the flow keeps changing and last sprint's recording is already wrong. Meanwhile the suite walks that exact flow every night. The recording should be a by-product of the test, not a chore that competes with it.

Solution

```
Feature: Documentation reel

  Scenario: Booking happy path
    Given I start video recording
    And I am on "/"
    When I click "Book now"
    And I fill in "name" with "Sara"
    And I fill in "dates" with "2026-06-01 to 2026-06-05"
    And I press "Search"
    And I click "Reserve" in the "Sunset Suite" row
    And I press "Confirm"
    Then ".confirmation-number" should be visible within 10 seconds
    And I stop video recording
```

How It Works

`Given I start video recording` is the first step on purpose. It builds a recording-enabled context, and because it is first, there is no page state to lose, nothing has logged in, nothing has navigated yet. From there the scenario runs as any other: `And I am on "/"` settles the page, the form fills resolve labels first, and `Then ".confirmation-number" should be visible within 10 seconds` polls every 100ms up to the deadline rather than sleeping. `And I stop video recording` finalises the recording and flushes the `.webm` to disk. The file lands in `./videos` named by the `{datetime}.{feature_file}.{scenario}.{status}.{ext}` pattern, so a passing run produces something like `2026-06-12T14-30-00.documentation-reel.Booking-happy-path.passed.webm`, you can read what it is, when it ran, and that it passed, all from the listing.

Here is the thing worth internalising: the recording follows the same waits the test follows. Because webship-js never inserts static sleeps, every state-changing step ends with `smartSettle`, which waits for `window.__webshipAjaxCount` to hit zero and the DOM to go quiet for 250ms, the video shows the app at rest at each step, not a stuttering blur of half-loaded pages. The reel is watchable precisely because the test is honest about waiting.

At scale: The reason this beats hand-recording is not that it saves five minutes once. It is that the recording can never drift from reality. When you run the same documentation scenario across twenty client sites, each one re-records itself every night against the live build, the demo reel for project A and project B are always current, always reflecting the deployed flow, with zero manual upkeep. A PO preparing a sprint review pulls the latest `.webm` from the artefact bundle instead of asking QA to re-screen-record. Standardise on `WEBSHIP_VIDEO=on` for a dedicated “demo” tagset and the whole portfolio gets self-updating walkthroughs for free.

Pro Tip: Keep documentation scenarios in their own feature file (or behind a `@demo` tag) and run them separately from your regression suite. They are slower by design, you may even slow them down deliberately (Recipe 522), and you do not want that wall-time tax on the suite that gates your releases.

Recipe 512 Recording With a Custom Name

Problem

A customer reports a checkout bug, you reproduce it, and you want to hand the engineer a video they can watch, not a wall of text. The default `{datetime}...{scenario}.{status}` filename is great for a directory you scan, but when you are attaching one file to one Jira ticket, you want the ticket ID right there in the name so it never gets lost in a `videos/` folder full of timestamped runs.

Solution

```
Scenario: Repro for ticket BUG-1284
  Given I start video recording
  And I am on "/checkout"
  When I click "Apply" without entering a coupon
  And I save the current video as "BUG-1284-empty-coupon.webm"
```

How It Works

`When I save the current video as "name"` waits for the active recording to finalise, then moves the finished `.webm` to the filename you chose. It overrides the automatic `{datetime}.{feature_file}.{scenario}.{status}` pattern for this one file only, every other recording in the directory keeps its default name and is left untouched. Drop the result into the bug-tracker ticket and the engineer has a perfect repro to work from.

Mind the ordering, and remember the warning from the chapter opener: `Given I start video recording` is the first step here because it rebuilds the context. If you had logged in or navigated before it, that state would be discarded and the recording would start from a blank page. Start first, then navigate. The save step is the natural last step, once you call it, the recording is closed and finalised.

Pro Tip: Parameterise the name. If your runner injects the ticket ID as an env var or a scenario tag, an After hook can build `${ticketId}-${status}.webm` automatically (the mechanics are in Recipe 518). Then anyone reproducing a bug just tags the scenario and the evidence files itself under the right ID.

Recipe 513 Recording Only on Failure

Problem

Your nightly run has 500 scenarios. Recording every one produces 5GB of `.webm` per run, most of it footage of tests passing, which nobody ever watches. But the one scenario that fails intermittently on the CI runner, the one you cannot reproduce on your laptop, is exactly the footage you would give a week of your life for. You want video for failures and nothing else.

Solution

The simplest path is the built-in mode, set `WEBSHIP_VIDEO=on-failure` and webship-js records every scenario but keeps only the ones that fail, deleting the passing recordings for you. When you want control over the kept filename, do it explicitly with hooks:

```
const { Before, After } = require('@cucumber/cucumber');
const fs = require('fs');

Before(async function(scenario) {
  this.videoEnabled = true;
  await this.startVideoRecording();
});

After(async function(scenario) {
  const videoPath = await this.stopVideoRecording();
  if (scenario.result.status === 'PASSED') {
    fs.unlinkSync(videoPath);   // discard on success
  } else {
    const safe = scenario.pickle.name.replace(/[^a-z0-9]/gi, '_');
    fs.renameSync(videoPath, `videos/FAIL_${safe}.webm`);
  }
});
```

How It Works

`WEBSHIP_VIDEO=on-failure` is the mode to reach for first, it records into the per-scenario context from birth (so no page state is ever lost) and, in its After hook, inspects `scenario.result.status` and unlinks the file on `PASSED`. The hook version above does the same dance manually, which is what you want when you need a custom name like `FAIL_<scenario>.webm` rather than the default pattern. Note the trade-off the env mode hides: recording everything and discarding the passes still pays the 5-10% wall-time cost on *every* scenario, because Playwright must record before it knows the outcome. You cannot record only failures without recording everything first, there is no retroactive capture, as the chapter opener warned. The saving is in storage and signal, not in runtime.

At scale: This is the single most valuable video setting for a team running many suites, so make it the default and stop thinking about it. Bake `WEBSHIP_VIDEO=on-failure` into the CI environment for the whole portfolio, one variable in the platform config, not a code change in forty repos. Every project's nightly run then yields a `videos/` folder that contains only failures, which collapses triage from "watch everything" to "watch the three that broke." Wire that folder into the artefact upload with `if: always()` and a QA lead can open last night's failures for any project without re-running a thing. The cross-project win is uniformity: when every suite keeps failure videos under the same convention, your PMO dashboard can link straight to the evidence for a red build, and "is release X safe?" stops being a question of trust and becomes a folder someone can open.

Pro Tip: Pair this with the screenshot-on-failure default (`WEBSHIP_SCREENSHOT_ON_FAILED` is `true` out of the box). A failure then leaves both a final-state PNG and a video of how it got there, the snapshot answers "what broke" in one glance, the video answers "why."

Recipe 514 Recording Only With a Tag

Problem

Most of your scenarios do not need video, they pass, nobody watches them, and the 5-10% wall-time cost adds up across a suite. But a handful are worth recording every single run: the flaky one you are still chasing, the payment flow legal wants on file, the smoke test a PO checks before sign-off. You want those recorded on every run and the rest left alone, decided in the feature file by whoever writes the scenario, not buried in a config someone forgets exists.

Solution

The built-in way needs no code at all, set `WEBSHIP_VIDEO=tag` and tag the scenarios you care about:

```
@video
Scenario: Reproduce ticket 1284
  Given I am on "/checkout"
  ...
```

If you want a different tag name or extra logic, write your own hook pair:

```
Before({ tags: '@record' }, async function() {
  await this.startVideoRecording();
});

After({ tags: '@record' }, async function() {
  await this.stopVideoRecording();
});
```

```
@record
Scenario: Reproduce ticket 1284
  Given I am on "/checkout"
  ...
```

How It Works

`WEBSHIP_VIDEO=tag` tells webship-js to build a recording-enabled context only for scenarios carrying `@video`, and to leave everything else on a plain context, so untagged scenarios skip the overhead entirely. Because the decision is made when the context is created, the tagged scenario records from its first step with no page state lost; you do not call the mid-scenario start step, so the warning from the opener does not apply here. The custom `@record` hook above is the same idea wired by hand: Cucumber runs the `Before` only for scenarios matching the tag expression, so `startVideoRecording` fires before any of your steps. Two notes on the built-in tags worth keeping straight, `@video` forces recording *on* even when the mode is `off`, and `@no-video` forces it *off* even when the mode is `on`. So `WEBSHIP_VIDEO=on` plus `@no-video` on three sensitive scenarios is the inverse pattern: record everything except the ones you opt out of.

At scale: Tag conventions are how a multi-project team keeps video sane without per-project config. Agree on one vocabulary, `@video` for “always record this”, `@no-video` for “never record this, it is sensitive”, and apply it identically across every suite. New hires learn it once and it reads the same in every repo. A product owner who cannot edit a hook file *can* add `@video` to the acceptance scenario they care about, which means the people writing the criteria control the evidence without filing a ticket to engineering. Run with `WEBSHIP_VIDEO=tag` as the portfolio default and your storage bill tracks the handful of tagged scenarios, not the thousands that pass quietly every night.

Pro Tip: Reserve `@no-video` for scenarios that touch PII or payment data even when your default mode is `off`. A future engineer who flips the suite to `WEBSHIP_VIDEO=on` for a debugging session will then automatically skip recording the sensitive flows, the safety is in the tag, not in remembering the rule.

Recipe 515 Custom Video Size

Problem

Your CI artefact bucket has a quota, and the platform team just sent the “your storage is at 90%” email. The 1280x720 default is crisp, but you are keeping failure videos across forty nightly suites and the maths does not work. Drop to 800x600 and you roughly halve the bytes per recording, still legible enough to watch a flake, small enough that the bucket stops filling.

Solution

Set the context options in your World:

```
// PlaywrightWorld.js
this.context = await browser.newContext({
  recordVideo: {
    dir: 'videos/',
    size: { width: 800, height: 600 },
  },
});
```

How It Works

`recordVideo.size` is independent of the browser viewport. Playwright records the page at the actual viewport, then scales each frame down to the size you asked for before encoding, so the layout is unchanged, only the resolution of the saved `.webm` drops. The file-size table later in this chapter puts numbers on it: 1280x720 runs 1-3 MB per minute, 800x600 runs 0.4-1 MB, and 640x480 runs 0.2-0.5 MB. Halving each dimension does not exactly halve the file, video compression is non-linear, but the saving is real and consistent. Below 640x480 the trade tips the wrong way: error toasts and form labels turn to mush and the recording stops being useful for the one job it has.

At scale: Resolution is a fleet-wide knob, so set it once and centrally. Rather than editing `recordVideo.size` in forty `PlaywrightWorld.js` files, prefer the env-var surface (`WEBSHIP_VIDEO_DIR` and the `WEBSHIP_VIDEO` mode) and treat size as a deliberate, documented override in projects that need it. A practical pattern for a portfolio: record CI failures at 800x600 to keep the nightly storage bill flat, and reserve 1280x720 for the `@demo` scenarios a PO or designer will actually present. Multiply 0.5 MB saved per minute across 200 failure videos a night across many projects and the platform team's quota email stops arriving.

Pro Tip: Decide resolution by audience, not by habit. Forensic failure footage only needs to be legible to an engineer scrubbing for the broken step, 800x600 is plenty. Footage going in front of a client or a sprint review earns the full 1280x720. Tag-gate the two cases so each scenario gets the size its viewers need.

Recipe 516 Recording at the Viewport Size

Problem

A bug only shows up on mobile, the menu overlaps the logo at the small breakpoint, but your videos all record at a fixed 1280x720, so the recording shows a desktop layout that looks fine. The footage and the bug disagree, and the engineer watching it cannot see what the user saw. You want the video to match the exact viewport the test ran at, so a mobile test produces mobile-shaped footage.

Solution

Omit `size` and Playwright records at the page's viewport size:

```
this.context = await browser.newContext({
  viewport: { width: 1366, height: 768 },
  recordVideo: { dir: 'videos/' },   // size omitted -> uses viewport
});
```

How It Works

When `recordVideo.size` is absent, Playwright falls back to the context's viewport for the recording dimensions, so the `.webm` is shaped exactly like the page under test. Pair this with webship-js's breakpoint registry and you get per-breakpoint recordings without thinking about it. The registry ships seven named widths, `xs` 375x667, `sm` 576x800, `md` 768x1024, `lg` 992x768, `xl` 1200x900 (the default), `xxl` 1400x900, and `xxxl` 1920x1080, so a scenario that runs at `xs` records a 375-wide vertical video that looks like a phone, while the same scenario at `xl` records a wide desktop frame. The footage matches the failure because it matches the viewport that produced it. (Override the registry per environment with the `WEBSHIP_SELECTORS_BREAKPOINTS` JSON if your design system uses different widths.)

At scale: Responsive coverage is where this earns its keep across a portfolio. If your suites run the same flows at multiple breakpoints, and for a multi-site team they should, viewport-sized recordings give you a visual record at every width, so a mobile-only regression on project C is obvious in its own footage instead of hidden behind a desktop frame. Standardise the breakpoint names across projects (the same `xs` / `sm` / `md` vocabulary everywhere) and a reviewer can pull "the mobile video for the checkout failure" on any site and know exactly what they are getting. One caveat for the storage-conscious: viewport recordings vary in size, and a 1920x1080 `xxxl` run is heavier than a fixed 800x600, so reserve full-viewport recording for the breakpoints where layout actually matters rather than every width on every project.

Recipe 517 Saving Videos to a Custom Directory

Problem

Your CI job uploads `tests/reports/` as the build artefact, and a reviewer downloads it expecting everything they need. But the videos land in a top-level `videos/` folder that the upload step never touches, so the report is there, the failure videos are not, and someone has to go re-run the job locally to watch the flake. You want the videos to live inside the evidence tree that already ships, so one archive holds the report, the screenshots, and the recordings together.

Solution

Point the directory at your evidence tree, via the env var when you can, in the World options when you cannot:

```
WEBSHIP_VIDEO_DIR=tests/evidence/videos npm test
```

```
this.context = await browser.newContext({
  recordVideo: { dir: 'tests/evidence/videos/' },
});
```

Then wire that one tree into the artefact upload alongside `tests/reports/`:

```
- uses: actions/upload-artifact@v4
  if: always()
  with:
    name: test-evidence
    path: tests/evidence/
```

How It Works

`WEBSHIP_VIDEO_DIR` is the supported way to relocate recordings, it defaults to `./videos`, and webship-js creates the directory if it does not exist, so you do not need a `mkdir` step. Setting it to a path under `tests/evidence/` means recording, report, and screenshots share one root. The `if: always()` on the upload is the load-bearing detail: without it, a failed test step aborts the job before the upload runs, and the failure footage, the only footage you actually wanted, never leaves the runner. With it, the archive uploads on pass or fail, so red builds keep their evidence. One download, and the reviewer has the cucumber HTML report, the failure PNGs, and the `.webm` recordings in a single tree.

At scale: A single evidence root per project is what makes cross-project triage bearable. When every suite in the portfolio writes report, screenshots, and video under the same `tests/evidence/` layout, the artefact-collection step in your CI templates is identical everywhere, copy one `upload-artifact` block into forty pipelines and never special-case a project. Set

`WEBSHIP_VIDEO_DIR` in the shared CI environment rather than hardcoding the path in each `PlaywrightWorld.js`, and the layout stays uniform even as projects come and go. A PMO that knows "the evidence bundle is always the `test-evidence` artefact, same shape on every job" can point a release-readiness dashboard at it without learning each project's quirks.

Pro Tip: Keep the evidence root *outside* your source tree's committed paths and in `.gitignore`. Recordings are build outputs, not source, they belong in CI artefacts and object storage (Recipe 523), never in the repo where they bloat clones and leak whatever was on screen.

Recipe 518 Per-Scenario Video Path

Problem

Three months from now, someone opens your video bucket looking for "the smoke-test login failure from mid-May" and finds two thousand files. The default `{datetime}.{feature_file}.{scenario}.{status}` pattern is good, but your team has its own convention, tag first so smoke and regression sort apart, then status so failures group, then date, then scenario. You want every recording to follow that shape so the bucket is searchable by the names your team actually thinks in.

Solution

Write an After hook with the full naming logic:

```
After(async function(scenario) {
  const stat = scenario.result.status.toLowerCase();
  const date = new Date().toISOString().slice(0, 10);
  const name = scenario.pickle.name.replace(/[^a-z0-9]/gi, '-
').toLowerCase();
  const tags = scenario.pickle.tags.map(t => t.name.replace(/^@/,
'')).join('_');
  const safe = `${tags}_${stat}_${date}_${name}.webm`;
  await this.saveVideoAs(safe);
});
```

How It Works

Cucumber hands the After hook a rich scenario object, and this recipe mines four parts of it. `scenario.result.status` is the outcome (`PASSED`, `FAILED`, `SKIPPED`), lowercased here so failures sort predictably. `scenario.pickle.name` is the human scenario title, slugged to filename-safe characters. `scenario.pickle.tags` is the full tag list, with the leading `@` stripped and joined so `@smoke @auth` becomes `smoke_auth`. Stitched together you get `smoke_auth_failed_2026-05-14_user-login.webm`, self-describing from the listing alone. `saveVideoAs` then finalises the active recording under that name, the same step Recipe 512 used inline, just driven from a hook so it applies to every scenario without touching the feature files.

This is the manual version of the built-in `{datetime}.{feature_file}.{scenario}.{status}.{ext}` pattern. Reach for the hook only when your team's sort order differs from the default, if the shipped pattern already sorts the way you think, use it and write no code.

At scale: A naming convention is worthless if it lives in forty different hook files that drift apart over time. Put this After hook in a shared step-definition package that every project depends on, so the recording names are byte-for-byte identical across the portfolio. Then a cross-project script, or a PMO dashboard, can parse any project's filenames the same way: split on `_`, read tag/status/date/scenario, and build a quality view that spans every suite. Uniform names are what let "show me every failed smoke-test video from last week, all projects" be a `grep`, not an archaeology dig. The moment each team invents its own scheme, that query dies.

Pro Tip: Append the CI run ID to the name in CI (`process.env.GITHUB_RUN_ID` or your provider's equivalent). When the same scenario fails on three consecutive nightly runs, run-ID-stamped filenames keep the three recordings distinct instead of overwriting each other, and you can watch the flake's history rather than just its latest instance.

Recipe 519 Printing the Video Path

Problem

You are debugging on your laptop, the scenario just ran, and you want to watch the recording right now, but you have not wired up artefact collection locally and the `{datetime}...` filename is a mouthful to guess. You do not want to `ls -t videos/` and eyeball the newest file. You want the test to just tell you where it put the recording.

Solution

```
Scenario: Locate the video
  Given I start video recording
  And I am on "/"
  When I press "Search"
  Then print video path
```

How It Works

`Then print video path` is a debug step in the same family as `Then print current URL` and `Then print API response`, it writes to the test stdout and changes nothing about the run. It resolves the active recording's destination and logs the absolute path, so you can copy it straight into a player (or `Cmd`/`Ctrl`-click it in a terminal that linkifies paths). It is a local-debugging convenience; in CI you would rely on the filename pattern and the artefact upload instead of reading stdout.

One catch tied to the chapter's recurring warning: `Given I start video recording` here rebuilds the context as its first step, so it must come before `And I am on "/"`. Print steps are safe anywhere, they only read, but the start step is not, so keep it first.

Pro Tip: Leave a `Then print video path` in a scenario you are actively debugging and remove it once the bug is fixed. It is noise in a clean CI log, but during a chase it saves you the `ls` dance every single run. Treat it like a `console.log`, handy while hunting, gone before you

commit.

Recipe 520 Combining Video With Screenshots

Problem

A scenario fails at “Place Order” and you have the video, so you can watch the whole run, but to compare the form state before submit against the error state after, you are scrubbing the timeline frame by frame, pausing, squinting. Video is the right tool for “what led here?” and the wrong tool for “what did the page look like at exactly this moment?”. You want both: a reel of the journey and crisp stills at the moments that matter.

Solution

```
Scenario: Hybrid evidence trail
  Given I start video recording
  And I am on "/checkout"
  When I fill in "cardNumber" with "4242424242424242"
  And I save screenshot with name "before-submit"
  And I press "Place Order"
  Then ".confirmation-number" should be visible within 10 seconds
  And I save screenshot with name "after-submit"
  And I stop video recording
```

How It Works

Two evidence systems run side by side and do not interfere. The video recorder streams the entire context at the context level; the screenshot steps capture single frames at the page level, written under `WEBSHIP_SCREENSHOT_DIR` (default `./screenshots`) by their own `{datetime}.{feature_file}.feature_{step_line}.{ext}` pattern. `And I save screenshot with name "before-submit"` drops a still right after the card field is filled and settled, and the matching `after-submit` shot captures the confirmed state. Because every state-changing step ends with `smartSettle`, both screenshots land on a quiet page rather than mid-transition, the still is sharp, not a blur of a half-rendered modal. The result is a layered trail: stills are the bookmarks, the video is the tape between them.

At scale: Two channels of evidence make a failure legible to two audiences at once. An engineer scrubs the video to find the broken step; a PMO or PO who will never open a `.webm` reads the before/after PNGs in the HTML report and understands the failure in one glance. Standardise both across the portfolio, `WEBSHIP_SCREENSHOT_ON_FAILED=true` (the default) for the automatic final-state still plus `WEBSHIP_VIDEO=on-failure` for the reel, and every red build in every project arrives with the same dual evidence, no per-scenario authoring required. The named-screenshot steps above are for the handful of flows where you want bookmarks at specific moments, not the default failure capture.

Pro Tip: Name screenshots by the state they capture (`before-submit`, `after-submit`), never by step number. Step numbers shift the moment someone edits the scenario; state names stay meaningful through every refactor and read clearly in the report months later.

Recipe 521 Recording in Headless and Headed Modes

Problem

A new engineer asks the reasonable question: “Do I get video when I run headed on my machine, or only in CI?” Behind it is a real worry, that the recording is a CI-only feature and local headed runs are a blind spot. You want one answer you can give every new hire: recording works the same

in both modes, so the footage is identical whether the browser was visible or not.

Solution

Playwright records identically in both modes. No special configuration. The same scenario:

```
# Headless (CI default)
npm test

# Headed (local) -- HEADLESS=false opens a visible browser
HEADLESS=false npm test
```

Produces the same `.webm` file. The difference: in headed mode, the human can also watch the browser live. The video is the playback for everyone who couldn't.

How It Works

Video recording is a context property, decided when the context is built, it has nothing to do with whether the browser draws to a screen. So `HEADLESS=false` (the supported switch; `playwright.config.ts` reads `process.env.HEADLESS !== 'false'`, and `npm run test:headed` sets it for you) changes only whether *you* see the window, not whether Playwright records it. There is one quiet difference baked into `playwright.config.ts` that affects the recording's pacing, though: `SLOW_MO` defaults to 300ms in headed mode versus 0 in CI. A headed run therefore moves a touch slower and its video plays a touch slower, which is usually a feature when you are watching to debug. The frames are the same; only the cadence shifts.

Pro Tip: When CI reports a failure you cannot see in the artefact video clearly, re-run that one scenario locally with `HEADLESS=false SLOW_MO=800`, the canonical webship-js debugging incantation. You then watch it live *and* keep a slow-motion `.webm` of the same run, doubling your odds of catching the moment it breaks.

Recipe 522 Slowing Down for the Camera

Problem

You record the booking flow for the sprint review, play it back, and it is a blur, forms fill instantly, the page jumps to confirmation, and a watcher who blinks misses the whole thing. The test is fast because it should be fast, but a documentation reel needs to be *followable*. You want the recording to slow to a human pace without slowing the suite that gates your releases.

Solution

The supported knob is the `SLOW_MO` env var (`playwright.config.ts` reads it directly), so you can pace a documentation run without touching code:

```
SLOW_MO=250 HEADLESS=false npm test    # 250ms between actions
```

If you build the browser yourself in a World, the equivalent launch option is `slowMo`:

```
this.browser = await chromium.launch({
  headless: false,
  slowMo: 250,   // 250ms delay between actions
});
```

How It Works

`slowMo` inserts a fixed delay before each Playwright action, click, fill, press, so every interaction is spaced out and the recorder captures a watchable cadence instead of instant jumps. It is a pacing layer on top of the waits, not a replacement for them: `smartSettle` still does the real synchronisation (ajax to zero, DOM quiet), and `slowMo` just adds breathing room a human eye can track. Worth knowing about the defaults baked into `playwright.config.ts`: `SLOW_MO` is 0 in CI, 300ms in headed mode, and 800ms when both `HEADLESS=false` and no explicit value are set, so a plain headed run is already gentler than CI. For documentation, 250-500ms is the sweet spot: fast enough to hold attention, slow enough to follow each click.

For CI, never set `SLOW_MO` above 0, it adds wall time to *every* action of *every* scenario, and across a nightly portfolio that compounds into real runner cost for zero testing benefit. Slow motion is a documentation tool, full stop.

At scale: Keep the slow, headed, recorded runs in their own lane, a separate `@demo` profile or a manually triggered pipeline, and never let `SLOW_MO` leak into the env your regression suites read. The failure mode at scale is a stray `SLOW_MO=300` in a shared CI variable silently taxing forty nightly suites; one engineer added it to debug, forgot to remove it, and now every run is 30% slower. Treat documentation pacing as an opt-in override per run, set on the command line, never persisted into the shared environment.

Pro Tip: Pair `SLOW_MO=400` with viewport-sized recording (Recipe 516) and a `@demo` tag for review reels. You get slow, correctly-sized, self-updating footage of every documented flow, and because it is tag-gated, it never touches the speed of the suite that actually guards your releases.

Recipe 523 Storing Videos to S3 or GCS

Problem

A flake fails one night in three, and you finally sit down to investigate, only to find the CI artefact retention is 30 days and the runs you needed expired last week. The footage that would have shown the pattern is gone. For evidence you might revisit months later, an intermittent failure, a

regression a customer reports against an old release, 30-day artefacts are not enough. You want failure videos in object storage that outlives the CI job that produced them.

Solution

Add a CI step that uploads after the test step:

```
- name: Upload videos to S3 on failure
  if: failure()
  run: |
    aws s3 sync videos/ s3://my-test-evidence/$(date +%Y-%m-%d)/${{
github.run_id }}/videos/
```

Or for GCS:

```
- name: Upload videos to GCS on failure
  if: failure()
  run: |
    gsutil -m rsync -r videos/ gs://my-test-evidence/$(date +%Y-%m-%d)/${{
github.run_id }}/
```

How It Works

`if: failure()` is the deliberate choice here, and it is the opposite of the `if: always()` you would use for the artefact bundle. You only push to long-term storage when the run failed, passing runs leave nothing behind, so the bucket fills with signal, not noise. The path partitions by date *and* `github.run_id`, so every failed run gets its own isolated prefix and nothing ever overwrites earlier evidence, the run that failed last Tuesday and the run that failed tonight sit in separate folders forever. `aws s3 sync` and `gsutil rsync` both upload only what changed, so the step is cheap even when `videos/` already holds a few files. The bucket becomes a permanent forensic library; link to the run's prefix straight from your bug tracker.

At scale: This is where video stops being a debugging toy and becomes an organisational asset. One bucket, partitioned `s3://evidence/<project>/<date>/<run-id>/`, holds failure footage for every project in the portfolio, so when a customer reports a regression against a release from

two months ago, you open the prefix for that project and date and watch what the suite saw, long after the CI artefacts expired. Set a lifecycle policy on the bucket (expire after 90 or 180 days) so storage cost stays bounded without manual cleanup. The DevOps win is that the upload step is identical across pipelines, only the `<project>` segment changes, so it drops into your shared CI template once. The PMO win is a single, durable place where "show me the evidence for that failed release" is always answerable.

Pro Tip: Print the bucket URL into the CI log right after the upload (`echo "Evidence: https://...$run_id/"`). A reviewer reading the failed job then has a one-click link to the footage instead of reconstructing the path by hand from the date and run ID.

Recipe 524 Cleaning Up Old Videos

Problem

Your laptop's disk warning pops up mid-sprint, and the culprit is a `videos/` directory you forgot existed, 20GB of `.webm` from a month of local runs, almost all of it footage of tests that passed and you never watched. Recordings are build output, not history; on your own machine you want them to clean themselves up so they never become the thing that fills your disk.

Solution

Add a pre-test cleanup:

```
# package.json
"scripts": {
  "pretest": "rm -rf videos/*",
  "test": "cucumber-js"
}
```

Or with retention:

```
"pretest": "find videos/ -mtime +7 -delete"
```

How It Works

`pretest` is an npm lifecycle script, npm runs it automatically right before `test`, so the cleanup happens every run without you remembering to invoke it. The first form wipes `videos/` clean before each run, so you only ever keep the recordings from your latest run; the second keeps anything modified in the last 7 days (`-mtime +7` matches files older than seven days) and deletes the rest, giving you a rolling local window. This mirrors the purge knob the screenshot system exposes (`WEBSHIP_SCREENSHOT_PURGE`), same idea, applied to video. Pick the form that matches how you work: wipe-every-run if you only ever care about the run you just did, rolling-window if you sometimes look back a few days.

A word of caution, this is a *local* convenience, not a CI pattern. In CI you do not purge; you upload (Recipe 517) and archive failures to object storage (Recipe 523). Deleting in CI would throw away the evidence the whole pipeline exists to capture.

Pro Tip: Keep CI and local retention deliberately different. CI keeps failure footage for as long as your bucket's lifecycle policy allows; your laptop keeps a 7-day rolling window and no more. Conflating the two, purging in CI, or hoarding locally, is how teams either lose evidence they needed or drown in footage they never wanted.

Recipe 525 Auto-Embedding Videos in the HTML Report

Problem

You hand the HTML report to a product owner so they can sign off on a release, and the report says "Booking happy path, failed." Their next message is "can you send me the video?", and now you are digging through `videos/`, finding the file, attaching it to a chat. Every failure becomes a manual evidence-fetch. You want the recording to play *inside* the report, so opening it shows both the result and the footage in one place, no follow-up needed.

Solution

Use a custom reporter:

```
// formatters/video-embedder.js
const { Formatter } = require('@cucumber/cucumber');
const fs = require('fs');

class VideoEmbedder extends Formatter {
  constructor(options) {
    super(options);
    options.eventBroadcaster.on('envelope', (envelope) => {
      if (envelope.testCaseFinished) {
        const videoPath = this.findVideoForCase(envelope);
        if (videoPath && fs.existsSync(videoPath)) {
          this.log(`<video controls width="800"><source src="${videoPath}"
type="video/webm"></video>\n`);
        }
      }
    });
  }
  // ...
}
module.exports = VideoEmbedder;
```

Wire it in `cucumber.js`:

```
module.exports = {
  default: {
    format: [
      'html:tests/reports/cucumber_report.html',
      './formatters/video-embedder.js:tests/reports/videos.html',
    ],
  },
};
```

How It Works

A Cucumber formatter listens to the run's event stream, the `envelope` events Cucumber broadcasts as it executes. This one watches for `testCaseFinished`, looks up the recording that belongs to the case that just ended, and, if the `.webm` exists, writes a `<video controls>` tag into its output file. Registering it as a second `format` entry in `cucumber.js` means it runs alongside the standard `html:` reporter without disturbing it, you get the normal report plus a videos page. This is a custom formatter sitting beside the report toolchain that `bin/generate-reports.js` drives; it does not replace the cucumber-html-reporter output, it augments it.

The closing caveat about lazy-loading is not optional at any real size: a naive embed puts every `<video src>` on one page, and opening it makes the browser fetch all of them at once. For a 200-scenario suite that is a multi-gigabyte page load that hangs the tab. Use `preload="none"` on each `<video>`, or render a poster thumbnail that loads the real file only on click, so the report opens instantly and streams each recording on demand.

At scale: The point of an embedded-video report is to put evidence where non-engineers already look. A PO or a PMO opens the HTML report, the artefact they already trust, and the failure footage is right there under the failed scenario, no "can you send me the video?" round-trip. Across a portfolio, ship the same formatter in your shared reporting setup so every project's report behaves identically: same layout, same inline playback, same place to click. That uniformity is what lets sign-off scale, a delivery manager reviewing ten projects' reports before a release learns one report format, not ten, and reads the evidence themselves instead of trusting a green checkmark. Combined with the metadata flags on `bin/generate-reports.js` (`--app-version`, `--env`, `--browser`), each report becomes a self-contained, watchable record of exactly what shipped where.

Pro Tip: Embed only failure videos, not every recording. A report with 200 passing-scenario videos is heavy and pointless, nobody watches a pass. Filter the formatter to `testCaseFinished` events with a failed result, and the report stays light while still carrying the footage that actually matters.

Performance and Sizing

A few rules of thumb from real projects:

METRIC	TYPICAL VALUE
Overhead per scenario	5-10% wall time
File size per minute (1280x720)	1-3 MB
File size per minute (800x600)	0.4-1 MB
File size per minute (640x480)	0.2-0.5 MB
Storage per CI run (50 scenarios)	50-200 MB
Storage per month (10 runs/day)	15-60 GB

Pick a retention window. Pick a resolution. Tag-gate recordings if you don't need them on every run. The defaults are tuned for "record everything in CI, keep for 30 days", adjust if your shape is different.

When to Record and When Not To

Record when:

- You have flakes you can't reproduce locally
- You need design or product review evidence
- You ship to non-engineers who can't read a stack trace
- You're onboarding and want a reference recording

Don’t record when:

- The suite already runs fast enough and adds no signal
- Storage is constrained
- The page is sensitive (medical, financial, PII) and the video would create a compliance issue

For the third case, redact: blur fields containing PII before recording, or skip the video step on tagged scenarios.

Video recording is one of the cheapest visibility wins in webship-js, three Gherkin steps, no infra changes, a tenfold reduction in “I can’t reproduce” time. Use it liberally; tune retention conservatively; the team will thank you on the next flake.

Chapter 47: Lessons From Varbase

Testing logic and recipes by Rajab Natshah, drawn from the Varbase acceptance suite.

A lot of webship-js's design did not come from a whiteboard. It came from years of running automated acceptance tests against **Varbase**, the enterprise Drupal distribution maintained at Vardot. Varbase is a starter kit used by global non-profits, governments, universities, and media houses. Every release ships through a Cucumber-driven acceptance suite that has, over time, grown into one of the most-tested Drupal distributions in the world. The lessons in this chapter come from that suite, patterns we kept, patterns we ditched, and surprises that changed how we wrote later steps.

If you want to read the source, the test fixtures live at:

- `github.com/Vardot/varbase/tree/10.1.x/tests/features/varbase`
- `github.com/Vardot/varbase-project/tree/11.0.x/tests/features`
- `github.com/Vardot/varbase-project/blob/11.0.x/tests/step-definitions/varbase-step-definitions.js`

Reading them is the fastest tour of webship-js in production. Below are the lessons distilled into recipes.

Three Principles From the Practice

Much of how the Varbase suite is written traces back to a couple of rules of thumb Mohammed Razem pressed on the team early, and which the suite then proved out release after release:

> ***Test the scenario, not the configuration.*** *Assert what a user can do, not what a settings form contains, and do not walk the configuration checkbox by checkbox. Configuration drifts and gets refactored; the behaviour it is supposed to produce is the contract worth pinning. A scenario that logs in and creates an article survives a dozen config refactors that a "the checkbox is checked" assertion would break on.*

> ***One basic scenario exercises the whole stack.*** *A single plain path, log in, create, save, see it published, runs through the front-end, the back-end, and the configuration that wires them together, all in one pass. So a small number of basic scenarios covers what dozens of config-by-config checks would, and catches the breakages that actually ship, without an elaborate suite. Varbase was built on exactly this, which is why its suite stays fast enough to run on every release.*

These two show up throughout the recipes below.

How the Varbase Test Suite Is Organised

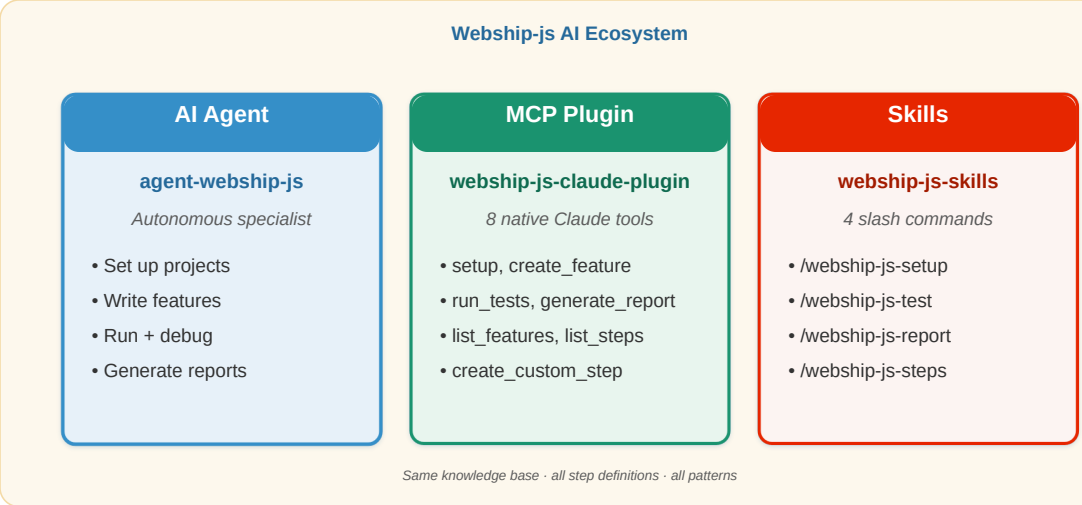


FIGURE 47.1 *How the Varbase Test Suite Is Organised*

Early Varbase shipped five numbered groups. A fresh Varbase 11 install has grown that into a **fifteen-group matrix**, still in the order a real installation builds itself up:

01-website-base-requirements	smoke tests, install profile, base config
02-user-management	login, password reset, role assignment
03-admin-pages	admin surfaces, reports, status
04-admin-users	user CRUD, masquerade, account flows
05-content-pages	landing and utility pages
06-content-blog	blog listing, post creation
07-content-contact	contact forms, submissions
08-content-homepage	homepage components and layout
09-drupal-canvas	the Canvas visual editor
10-content-permissions	per-role content access matrix
11-content-workflow	moderation states, transitions
12-content-access-and-lock	unpublished access, content lock
13-varbase-recipes	recipe-driven feature installs
14-ai	AI agents and assisted authoring
15-quality	accessibility, performance, audits

FIGURE 47.2 *How the Varbase Test Suite Is Organised*

Every scenario carries environment tags: `@local @development @staging @production`. The tag set determines where the scenario runs. The smoke tests run everywhere; the destructive admin tests run only on `@local` and `@development`.

The matrix grew because the product did, AI authoring, the Canvas editor, recipe installs, and a dedicated quality group are all newer than the original five, but the shape did not change: numbered groups, environment-aware tags, smoke-first ordering. That is the shape recommended in Chapter 19, and it comes from years of pain on Varbase. Other organisations were tried: by module, by author, one giant directory. Each collapsed under its own weight within a year. The numbered-group shape survived because it encodes two things at once, *dependency order* (groups run in the order a real installation builds itself up) and *blast radius* (a failure in `01-` means stop reading, everything downstream is noise). Adding a group is just appending the next number, so the suite scales without ever being reorganised, and anyone who has worked on one Varbase suite can navigate all fifteen groups without a map.

Recipe 526 Smoke First, Then Depth

Lesson

It is 11pm on release night and the full suite has been running for forty minutes when scenario 214 fails, because the install profile broke an hour ago and nothing was watching. Every failure after that is noise from the same root cause. Run the cheapest, broadest tests first. If the foundation broke, you want the first failure to be the actionable one, three minutes in, not the two-hundredth, forty minutes in.

Pattern from Varbase

The `01-website-base-requirements/` directory contains tests like “homepage loads”, “admin login page renders”, “system status report has zero errors”. They run in 30 seconds. If they fail, the rest of the suite is skipped.

```
@smoke @local @development @staging @production
Feature: Website base requirements

  Scenario: Homepage is reachable
    Given I am on "/"
    Then I should see "Welcome"
    And there should be no JavaScript errors

  Scenario: System status is clean
    Given I am a logged in user with the "Webmaster" user
    And I am on "/admin/reports/status"
    Then I should not see "Errors"
```

Recipe

Reserve the lowest-numbered group for tests that take seconds and cover assumptions every other test depends on. If those fail, abort early. Wire it via Cucumber's `--tags @smoke` flag on a fail-fast CI run before the full suite.

There is more happening in those two scenarios than the Gherkin shows. Every `Given I am on` navigation goes through webship-js's smartSettle, fetch/XHR counters at zero, no pending timers, DOM quiet for 250ms, so "homepage is reachable" really means "homepage loaded, finished its AJAX, and stopped mutating." And because webship-js watches the console on every step (`WEBSHIP_JS_ERROR_MODE` defaults to `warn`; flip it to `fail` for smoke runs, or tag the feature `@js-fail`), the smoke pass doubles as a JavaScript health check without a single extra step.

At scale: the smoke gate is where this pattern pays for itself. At Vardot the same `@smoke` convention runs across every project pipeline, so one CI template serves all of them: stage one runs `--tags @smoke` and kills the build in under five minutes if red; stage two runs the full suite only on green. Across dozens of nightly pipelines, that single gate is the difference between burning forty runner-minutes per broken site and burning four.

Recipe 527 Environment-Aware Tags

Lesson

A test that drops the database is fine on `@local`. The same test on `@production` would be a career-limiting move, and the scary part is how easy that move is: one wrong `LAUNCH_URL` in a CI variable and your destructive suite is pointed at the live site. Tag every scenario with the environments it is safe to run in, so the protection lives in the test itself instead of in everyone remembering to be careful.

Pattern from Varbase

```
@local @development
Scenario: Admin can disable a user
  Given I am a logged in user with the "Webmaster" user
  And I am on "/admin/people"
  When I click "Edit" in the "Normal user" row
  And I check "Blocked"
  And I press "Save"
  Then I should see "The changes have been saved"
```

This test never runs against staging or production, by construction. The reverse, read-only smoke tests, carry all four environment tags.

Recipe

For every new scenario, ask: which environments can this run in safely? Tag accordingly. Wire CI to filter by environment:

```
ENV=production cucumber-js --tags "@$ENV"
```

Two details make this fail safe rather than merely tidy. Tags are an allow-list: under a positive tag filter, a scenario carrying no environment tag never runs in any pipeline, so a forgotten tag means a skipped test, not a destructive one loose on production. And the target site comes from `LAUNCH_URL`, webship-js reads it into `worldParameters.launchUrl`, defaulting to `http://localhost:8080`, so one pipeline definition serves every environment by swapping two variables: `ENV` picks *which* scenarios run, `LAUNCH_URL` picks *where* they run. The pairing is the protection: the production pipeline physically cannot select a `@local`-only scenario, no matter whose finger slips.

At scale: this is how one suite serves a whole fleet. At Vardot the CI template is identical across projects; only `ENV` and `LAUNCH_URL` differ per pipeline, and both live in CI variables rather than in the repo. Nobody edits a config file to point tests at a new environment, which also means

nobody fat-fingers one. And when a security review asks “what stops the destructive suite from reaching production?”, the answer is a tag expression in version control, not a promise that everyone stays careful.

Recipe 528 The “Logged In As” Custom Step

Lesson

Open a feature file from the early Varbase suite and the first six lines of every scenario were identical: go to the login page, fill in the name, fill in the password, press the button, wait. Multiply by a few hundred scenarios and you are paying ten seconds of login per scenario and forcing every reviewer to re-read the same six lines until their eyes learn to skip them, and eyes that skip the setup also skip the one scenario where the setup is subtly wrong, which is the one hiding the bug.

Pattern from Varbase

Varbase ships a single custom step that handles the entire login flow:

```
Given(/^I am a logged in user with( the)*( username)* "([^"]*)?"(
user)*$/,
  async function(_, __, username, ___) {
    const user = USERS[username];
    await this.page.goto('/user/login');
    await this.page.fill('#edit-name', user.name);
    await this.page.fill('#edit-pass', user.pass);
    await this.page.click('#edit-submit');
    await this.page.waitForSelector('.toolbar-user');
  });
```

Then every feature uses one line:

```
Given I am a logged in user with the "Webmaster" user
```

Recipe

Build one custom step per long, repeated flow. Wrap the dance once; use the readable name everywhere. Login is the most common, but the same pattern applies to “I have a published article”, “I have an active reservation”, “the cart contains 3 items”. Each becomes a one-liner that reads like a domain sentence.

Two implementation details in the Varbase step are worth stealing. The regex’s optional groups (`( the)*`, `( username)*`) let one definition match several natural phrasings, so authors never have to memorise a single blessed wording. And the final `waitForSelector('.toolbar-user')` defines success as *an element that only exists when you are logged in*, the step waits for evidence, not for time. When login gets slower, the step still passes; when login silently fails, the step fails right there instead of three steps later in some unrelated assertion.

If even one scripted login per scenario is too slow, webship-js 2.0 ships auth-state steps that save and restore Playwright’s `storageState`: log in once, save the state, and every later scenario restores cookies and local storage in milliseconds. The custom step stays as the readable front door; the storage-state steps are the fast lane behind it.

At scale: this step is the reason the login line reads identically on every Vardot project. The definition ships in the shared package; only the fixture file mapping labels like “Webmaster” to real credentials differs per site. A QA engineer who rotates across three projects in a week types the same sentence on all three, and when one client adds a 2FA screen, the fix lands in one step file, not in forty copies of a login dance.

Recipe 529 Test User Fixtures

Lesson

The staging password rotates on a Tuesday and forty scenarios go red at once. Someone greps the feature files and finds the old password hard-coded in thirty places, each pasted from the scenario before it. The fix eats an afternoon of find-and-replace, and the same afternoon repeats at the next rotation, because the data lives *in* the scenarios instead of behind them.

Pattern from Varbase

```
// fixtures/users.js
module.exports = {
  Webmaster: {
    name: 'webmaster',
    pass: process.env.WEBMASTER_PASS || 'dD.123123ddd',
    roles: ['administrator'],
  },
  'Normal user': {
    name: 'normal_user',
    pass: process.env.NORMAL_USER_PASS || 'dD.123123ddd',
    roles: ['authenticated'],
  },
  ...
};
```

The custom step from Recipe 528 looks up the user by label:

```
const user = USERS[username];
```

Recipe

Maintain a single fixture file for test data. Use environment variables for secrets. Reference fixtures by label in features. When a password changes, update one line. When a new test user is needed, add one record. Features never change.

Notice the shape of the fixture: the key is a human label (“Webmaster”, “Normal user”), the value is machine detail (account name, password, roles). Features speak only the label. That one level of indirection is what lets the suite survive credential rotation, environment differences, even a rename of the underlying account, the Gherkin sentence `I am a logged in user with the "Webmaster" user` outlives all of it. And the `process.env.WEBMASTER_PASS || 'dD.123123ddd'` pattern gives you a safe local default and a CI override in the same line: developers run against a known local password, pipelines inject the real secret, and no production credential ever lands in the repo.

At scale: standardise the labels, not the credentials. Every Vardot project has a “Webmaster”, a “Content editor”, and a “Normal user”, different real accounts on every site, same labels everywhere. That convention makes feature files portable between projects, lets onboarding docs say “use the Webmaster fixture” instead of circulating passwords, and gives the security team exactly one answer to “where do test credentials live?”: in each pipeline’s secret store, injected as env vars.

Recipe 530 Multi-Step Scenarios That Mirror Real Workflows

Lesson

The create form has a test. The edit form has a test. Both pass, and the bug ships anyway, because it only appears when an editor creates an article, then edits it, then unpublishes it, and the unpublish drops the revision the edit just made. The bugs that matter happen at workflow seams, and a suite built only from single-action scenarios never crosses a seam.

Pattern from Varbase

```
@content-management @local @development
Scenario: Article workflow end-to-end
  Given I am a logged in user with the "Content editor" user
  And I am on "/node/add/article"
  When I fill in the field "Title" with "My Article"
  And I fill in the WYSIWYG field "Body" with the "Lorem ipsum..."
  And I press "Save and publish"
  Then I should see "Article My Article has been created"
  When I click "Edit"
  And I fill in the field "Title" with "My Updated Article"
  And I press "Save and keep published"
  Then I should see "Article My Updated Article has been updated"
  When I click "Edit"
  And I select "Unpublished" from "Status"
  And I press "Save"
  Then I should see "has been updated"
  And the path should be "/node/X/edit"
```

The scenario is long, but it covers a real lifecycle: create → edit → unpublish.

Recipe

When in doubt, mirror what a human would do. A 20-step scenario that follows a user journey catches bugs three single-step scenarios would miss. The trade-off is failure isolation, when a 20-step scenario fails, you need the report to tell you which step. Webship-js solves that twice over: on failure it screenshots automatically (`WEBSHIP_SCREENSHOT_ON_FAILED` defaults to true), and after every state-changing step its AfterStep hook runs `smartSettle(page, 1500)`, so step 14 never fails because step 13's AJAX was still in flight. Long journeys are exactly where that auto-settle earns its keep, the failure you see is the failure that actually happened.

Pro Tip: tag your longest journeys `@screenshots`. That flips on every-step capture for just those scenarios, and when one fails you can flip through the screenshot trail like a flipbook and watch the workflow break in slow motion.

Recipe 531 The Background Block

Lesson

Five scenarios in a feature, and each one opens with the same two lines of setup. The duplication looks harmless until the admin path changes and the fix is a five-line diff that should have been one, or until a new scenario quietly gets the setup slightly wrong and spends six months testing a different page than its siblings.

Pattern from Varbase

```
Feature: Admin content management

  Background:
    Given I am a logged in user with the "Webmaster" user
    And I am on "/admin/content"

  Scenario: Filter by published status
    When I select "Published" from "Status"
    And I press "Filter"
    Then I should see at least one published article

  Scenario: Filter by author
    When I fill in "Authored by" with "webmaster"
    And I press "Filter"
    Then I should see articles authored by "webmaster"
```

`Background:` runs before every scenario in the file. Two scenarios, one shared setup, no repetition.

Recipe

When two or more scenarios in a feature share the first 2-3 steps, lift them into `Background:`. When they share more than 5 steps, consider whether they really belong in the same feature, or whether a feature split would read better.

Remember what `Background:` is not: it is not run-once setup. It executes before *every* scenario in the file, so a ten-scenario feature pays that login ten times. That is the correct default, scenarios stay independent and can run in any order, or in parallel, but if the repeated login starts to dominate runtime, pair the pattern with webship-js's auth-state steps: log in once, save the `storageState`, and let the Background restore it in milliseconds instead of replaying the form. Same isolation, a fraction of the cost.

Recipe 532 Tag-Driven Test Selection

Lesson

Your commit changes one stylesheet, and CI makes you wait forty minutes while 500 scenarios re-verify the password-reset flow. By Thursday, developers are merging without waiting for green, and once people stop waiting for the suite, the suite has stopped doing its job. A suite that can only run all-or-nothing will eventually run as nothing.

Pattern from Varbase

```
# Full suite (nightly)
cucumber-js

# Smoke only (every commit)
cucumber-js --tags @smoke

# JavaScript-dependent scenarios only (when JS changes)
cucumber-js --tags @javascript

# Skip flaky scenarios (during a bug hunt)
cucumber-js --tags "not @flaky"

# Run only a specific area (during local development)
cucumber-js --tags @content-management
```

Recipe

Adopt a tag taxonomy from day one. Suggested:

- Environment tags: `@local @development @staging @production`
- Speed tags: `@smoke @full @regression`
- Area tags: `@user-management @content @api ...`
- Status tags: `@flaky @wip @broken`

Wire each tag to a CI workflow. The result is a test suite that's responsive at every commit cadence, not all-or-nothing.

One thing the taxonomy gives you that is easy to miss: in webship-js, tags don't just *select* scenarios, they change how scenarios *run*. `@screenshots` switches on per-step capture, `@video` forces a recording, `@js-fail` makes any console error fail the scenario, and `@no-video` and `@js-off` switch those back off. So `cucumber-js --tags @flaky` doesn't just isolate your problem children, tag those same scenarios `@video @screenshots` and the next run comes back with full forensics, no config edits anywhere.

At scale: the taxonomy is only useful if it is the *same* taxonomy everywhere. Vardot fixed the tag names once, org-wide, and every project inherits them, which means one CI template serves every repo, a QA lead can ask “show me @smoke across all thirty sites” and get a real answer, and an engineer borrowed from another team already knows what `@flaky` means before they open the README. The worst version of tagging is each project inventing its own dialect.

Recipe 533 The Three-File Setup

Lesson

The third project you onboard is where layout drift starts to bite: the first repo keeps step definitions in `tests/steps/`, the second in `features/support/`, the third somewhere only its author remembers. The new hire’s first PR puts files in the wrong directory because there was no right one. A real Varbase test project has exactly three top-level pieces, and every project keeps them in the same place:

```
cucumber.js                    Cucumber runtime config
playwright.config.ts           Playwright runtime config
tests/
  step-definitions/
    *.steps.js                 shared step files
  features/
    01-*/
    02-*/
    …
  hooks/
    world.js                   shared World class
    setup.js                   before-all
    teardown.js                after-all
```

FIGURE 47.3 *Lesson*

Everything else is project-specific.

Recipe

Don't over-architect. Three top-level config files, one step-definitions directory, one features directory organised by area. That's it. Any extra directory layer past the second adds friction for new contributors.

This layout is not just convention, it is what webship-js's defaults already expect. The stock `cucumber.js` profile loads steps from `tests/step-definitions/**/*.js`, features from `tests/features/**/*.feature`, and writes the JSON report to `tests/reports/cucumber_report.json`, with a 45-second step timeout. The stock `playwright.config.ts` reads `BROWSER`, `HEADLESS`, and `SLOW_MO` from the environment. Keep the standard shape and your per-project configuration shrinks to a handful of env vars; fight the shape and every project grows a bespoke config somebody has to maintain forever.

At scale: identical layout is an onboarding feature. At Vardot, anyone who has touched one suite can open any other repo and know where the features, steps, and reports live before the clone finishes. Project differences are expressed in env vars, `LAUNCH_URL`, `BROWSER`, credentials, never in moved directories. When something genuinely must differ structurally, write down why, because that difference will tax every engineer who visits the repo after you.

Recipe 534 The Site-Status Sentinel

Lesson

The disk fills up on a Friday night, the files directory turns read-only, and nothing visibly breaks, pages render, login works, the suite stays green. Monday morning an editor tries to upload an image and the support queue lights up. Drupal's status report had been flagging the problem all weekend; nobody was looking at it. That report aggregates DB connectivity, file permissions, module health, and update availability into one page, which makes it one page a robot should read every night.

Pattern from Varbase

```
@smoke @local @development @staging @production
Scenario: System status report is clean
  Given I am a logged in user with the "Webmaster" user
  And I am on "/admin/reports/status"
  Then I should not see "Errors"
  And there should be no JavaScript errors
```

This single scenario catches an enormous class of bugs: a missing module file, a writable directory turning read-only, a database connection drifting, an expired SSL cert on a back-end service.

Recipe

Find the equivalent in your stack. WordPress: `/wp-admin/site-health.php`. Laravel: `/up`. Rails: `/rails/info/properties`. Express: a custom `/_health` endpoint. Add a smoke scenario that asserts the dashboard reports green. Run it as part of every CI run.

The scenario is two assertions, but each is wider than it looks. `I should not see "Errors"` reads the page the platform itself uses to summarise dozens of subsystem checks, so you inherit every check the platform adds in future versions, for free. And `there should be no JavaScript errors` rides on webship-js's console capture, which has been listening on every step all along (`WEBSHIP_JS_ERROR_MODE` defaults to `warn`); the step turns that quiet listening into a hard assertion. One scenario, dozens of failure modes covered.

At scale: one sentinel per site, run nightly, becomes a fleet health board. Generate the report with metadata, `bin/generate-reports.js --env staging --app-version 10.1.3`, and "are all thirty sites healthy this morning?" stops being thirty SSH sessions and becomes one glance down a list. This is the cheapest monitoring you will ever ship, because the check already existed inside the platform; you just pointed a scenario at it.

Recipe 535 The “It Doesn’t Crash” Scenario

Lesson

A deploy goes out with a broken template on the search page. Every test passes, because every test starts at the homepage and follows the happy paths, and none of them ever visits `/search`. The first person to find the white screen is a visitor, the second is your client, and the third, finally, is you. Half of acceptance testing is “this page renders without exploding.” Don’t underestimate it.

Pattern from Varbase

```
@smoke
Scenario Outline: Page <path> renders without errors
  Given I am on "<path>"
  Then there should be no JavaScript errors
  And I should not see "Drupal\\Core\\Database\\DatabaseExceptionWrapper"
  And I should not see "Fatal error"
  And I should not see "PHP Fatal Error"

  Examples:
    | path                  |
    | /                     |
    | /node                 |
    | /admin                |
    | /admin/structure      |
    | /admin/reports        |
    | /search               |
```

One outline, six paths, six guaranteed-rendering pages, and adding a seventh costs one table row.

Recipe

Write a smoke outline that visits every critical path. It's not exciting but it catches deploys that broke routing, asset compilation, or template parsing. Run it before any complex scenarios, if the basics fail, the rest is noise.

The outline earns more than it shows. Each `Given I am on "<path>"` runs through smartSettle, the page must reach `domcontentloaded`, finish its fetch/XHR traffic, drain its pending timers, and hold the DOM quiet for 250ms before the step passes, so "renders" means *finished rendering*, not "returned some HTML". The negative assertions then sweep the settled page for the signatures your stack prints when it dies. Swap in your own: Rails' "We're sorry, but something went wrong", Laravel's "Whoops", a React error boundary's fallback text. The error strings are the only framework-specific part of the recipe.

Recipe 536 The Permission Matrix

Lesson

A permissions refactor lands, and in review someone asks the simple question: "can a Content editor still reach /admin/people?" Nobody knows. Answering by hand means logging in as four different users and clicking through every admin path, so nobody does, and the regression ships. "Who can access what" is the most regression-prone area in any CMS precisely because checking it manually is tedious enough that humans skip it.

Pattern from Varbase

```
Scenario Outline: <role> access to <path>
  Given I am a logged in user with the "<role>" user
  When I am on "<path>"
  Then I should <expectation>

  Examples:
    | role            | path                | expectation        |
    | Webmaster       | /admin/people       | not see "Access denied" |
    | Site editor     | /admin/people       | see "Access denied"     |
    | Content editor  | /admin/people       | see "Access denied"     |
    | Normal user     | /admin/people       | see "Access denied"     |
    | Webmaster       | /admin/structure    | not see "Access denied" |
    | Site editor     | /admin/structure    | not see "Access denied" |
    | Content editor  | /admin/structure    | see "Access denied"     |
    | Normal user     | /admin/structure    | see "Access denied"     |
```

One outline, four roles, every admin path. A regression in `permissions.module` lights up immediately.

Recipe

Build a single Permission Matrix feature per role-aware area. Include both positive (allowed) and negative (denied) cases. Update the matrix whenever roles or permissions change. A 20-row table beats 20 separate scenarios for maintenance.

The negative rows matter more than the positive ones. A role that *gains* access you didn't intend is a security bug, and it is invisible to manual testing because nothing looks broken. The Examples table also turns the access policy into a reviewable artifact: a product owner can read it top to bottom and confirm it matches intent without reading a line of code, and a permissions change shows up in the diff as changed table rows, the clearest review a security-sensitive change can get.

At scale: keep the matrix shape identical across projects and it becomes your audit answer. When a government client asks "show us who can administer users", the response is the feature file plus its latest green report, the spec and the proof in one place. A new role on one site means new rows,

not new scenarios; the outline machinery is already there, and so is the habit of reading it.

Recipe 537 The Audit Trail Test

Lesson

The incident review is the worst possible moment to learn your audit log stopped recording two releases ago. Audit trails fail silently by nature, no user complains, no page breaks, the table just quietly stops growing. Sensitive systems need an audit trail, and the audit trail itself needs a test, because the day you consult it is the day you cannot afford it to be empty.

Pattern from Varbase

```
@audit @local @development
Scenario: Failed admin login is recorded
  Given I am on "/user/login"
  When I fill in "username" with "webmaster"
  And I fill in "password" with "wrong-password"
  And I press "Log in"
  Then I should see "Unrecognized username or password"

  When I am a logged in user with the "Webmaster" user
  And I am on "/admin/reports/audit-trail"
  Then I should see "Authentication failure"
  And I should see "webmaster" in the most recent audit row
```

The first half creates an audit event; the second half asserts the event was recorded.

Recipe

For every audit-worthy action in your app, write a paired test: action + verification. The act of writing the test forces you to confirm the audit pipeline is wired correctly. Skip the test and the audit log will silently break on the day you need it most.

Note the structure: the scenario *causes* an event as one identity, then *verifies* it as another. The failed login must come from a clean session, if a previous scenario could have left cookies behind, open with webship-js's `Given I am an anonymous user`, which clears cookies before navigating, so the failure you record is genuinely an anonymous one. Self-contained cause and effect, no dependence on whatever happened to be in the log already. That is what makes this a test instead of a hope.

Recipe 538 The JSON-API Contract Test

Lesson

The mobile team builds against your JSON-API. Six months later a module update renames a field, their app breaks on launch, and you find out from their bug report, filed after their users filed theirs. Nothing in your UI suite noticed, because your UI never reads that field. A REST API contract is a contract whether you wrote it or someone else did, and the side that can test it cheapest is yours.

Pattern from Varbase

```
@api @local @development @staging
Scenario: JSON-API exposes articles
  Given I am authenticating as "admin_api" with "API_TOKEN" password
  When I send a GET request to "/jsonapi/node/article"
  Then the API response code should be 200
  And the JSON response should have property "data"
  And the JSON response should have property "links.self"
  And the JSON response should not contain "errors"
```

Recipe

For every JSON-API resource, write a contract test that asserts: response code, the presence of `data`, the presence of canonical `links`, and the absence of `errors`. If the contract drifts, the test fails. Stakeholders who depend on the API get a heads-up before they file a bug.

The block above shows the Behat-era phrasing from the Varbase suite; webship-js 2.0 runs API scenarios through axios with no browser involved, which makes them the fastest tests you own. The modern shape: `Given the API base URL is "..."` to point at the service, `Given I set header "H" with value "V"` for the auth token, then `Then the API response code should be 200` plus JSON-pointer assertions like `Then the JSON response should have "/data" equal to value`, pointers reach into nested structures without brittle string matching. Either way, assert the *shape* (data, links, no errors), not the full payload: contract tests that pin every byte break on every harmless change, and tests that cry wolf get ignored.

At scale: contract tests are the cheapest cross-team insurance you can buy. Tag them `@api`, run them on every backend deploy, no browser, so they cost seconds, and publish the report where the consuming teams can see it. At Vardot, the JSON-API contract checks are what let a distribution update roll out to client sites without a round of “did this break your integration?” emails.

Recipe 539 The Tour Test

Lesson

A designer reorders the admin toolbar, ships it, and the product tour now anchors its third tooltip to empty space. No error fires, no test fails, no log line appears, a tour pointing at nothing is, to every machine involved, working as intended. You learn about it months later from a customer-success call, after every new user in between took a broken first-run experience as their introduction to your product.

Pattern from Varbase

```
@tour
Scenario: Admin tour runs end-to-end
  Given I am a logged in user with the "Webmaster" user
  When I am on "/admin"
  And I click "Tour"
  Then I should see "Welcome to Varbase" in the tour
  When I click next button in tour
  Then I should see "Manage content" in the tour
  When I click next button in tour
  Then I should see "Manage users" in the tour
  ...
```

Note `click next button in tour` is a custom step, it knows the tour-library's selector pattern.

Recipe

If your product ships a tour, add a tour test. The cost is low; the value spikes the day a designer re-orders the page and the tour starts pointing at the wrong element.

The custom step is the load-bearing decision here. `click next button in tour` hides the tour library's selector pattern inside exactly one step definition, so when you swap tour libraries, or the library renames its CSS classes, you fix one line in one file and every tour scenario keeps reading the same. And write the assertions against the tooltip *text* the user reads, not the library's DOM structure: the text is the part your product team actually owns.

Recipe 540 The Persistent-Login Test

Lesson

No developer manually tests “Remember me”, doing it honestly means closing the browser and coming back tomorrow, or thirty days from now. So the checkbox ships, the persistent cookie silently never gets set, and the bug lives for a quarter until a pattern surfaces in support tickets: “the site keeps logging me out.” Easy to write, impossible to test by clicking around. Exactly the kind of feature automation exists for.

Pattern from Varbase

```
@auth @local @development
Scenario: Persistent login survives session expiry
  Given I am on "/user/login"
  And I fill in "username" with "normal_user"
  And I fill in "password" with "dD.123123ddd"
  And I check "Remember me"
  And I press "Log in"
  Then I should see "Welcome"
  When the cookie "session" is removed
  And I reload the page
  Then I should still see "Welcome"   # persistent cookie holds the
session
```

Recipe

If “Remember me” exists, test it. The bug is usually that the persistent cookie wasn’t set, and only an automated test catches it because no developer manually waits 30 days to test their own login.

The trick in the scenario is simulating time instead of waiting for it: delete the short-lived session cookie, reload, and see whether the persistent token re-authenticates you. Make the test sharper by asserting the cookie exists *before* you rely on it, webship-js ships twelve cookie matchers, so

`Then a cookie with the name "session" should exist` (or a name-containing variant, for cookies with generated suffixes) confirms the persistent cookie was actually written at login. That splits any failure into two clean diagnoses: cookie never set, or cookie set but ignored on the way back in.

Recipe 541 The Password-Reset Flow

Lesson

A deploy swaps the mail backend, the password-reset email silently stops sending, and the users who hit the bug are by definition the ones who cannot log in to complain. They retry, fail, and leave. Password reset is the single most-tested feature in any auth system and the single most-broken after a deploy goes wrong, because it spans the app, the queue, and the mail service, and most suites only test the first third.

Pattern from Varbase

```
@auth @local @development
Scenario: Request a password reset
  Given I start recording network requests
  And I am on "/user/password"
  When I fill in "Username or email address" with "normal_user"
  And I press "Submit"
  Then I should see "Further instructions have been sent"
  And a POST request to "**/user/password" should have been made
```

For the email half, hook into a mail-catching service (MailHog locally) and assert the email arrived:

```
  When I am on "http://localhost:8025"
  Then I should see "Replacement login information for normal_user"
```

Recipe

Test the request half always. Test the email-receipt half whenever you have a local mail catcher. The full password-reset flow is one of the highest-value tests in any auth system.

Recipe 542 The Content-Editor Workflow

Lesson

WYSIWYG editors have a thousand small ways to break. The fix for one breaks two more. Test the editor as a workflow, not a widget.

Pattern from Varbase

```
@content-management
Scenario: Editor preserves content across save and edit
  Given I am a logged in user with the "Content editor" user
  And I am on "/node/add/article"
  When I fill in the field "Title" with "My Article"
  And I fill in the WYSIWYG field "Body" with the "Paragraph
one.\n\nParagraph two."
  And I press "Save and publish"
  Then I should see "My Article"
  And I should see "Paragraph one."
  And I should see "Paragraph two."
  When I click "Edit"
  Then the WYSIWYG field "Body" should contain "Paragraph one."
  And the WYSIWYG field "Body" should contain "Paragraph two."
```

The second half is the kicker: the editor must round-trip content through save and re-edit. Many editors lose newlines or convert paragraph tags on save.

Recipe

For every rich-text editor, write a round-trip test. The bug it catches is the one your customers complain about most.

Recipe 543 The Image-Upload Workflow

Lesson

File uploads sit at the seam between browser, app server, and storage backend. The seam leaks regularly.

Pattern from Varbase

```
@media @local @development
Scenario: Image upload generates derivatives
  Given I am a logged in user with the "Content editor" user
  And I am on "/media/add/image"
  When I attach the file "fixtures/test-image.jpg" to "Image"
  And I fill in "Name" with "Test Image"
  And I press "Save"
  Then I should see "Test Image"
  And the path should match "/media/[0-9]+"
  When I am on "/admin/content/media"
  Then I should see "Test Image"
  And I should see "test-image.jpg"
```

Followed by the derivatives check:

```
  When I am on "/sites/default/files/styles/large/public/test-image.jpg"
  Then the response status code should be 200
```

Recipe

Test uploads end-to-end. Don't stop at "the file uploaded", assert the derivative URLs serve correctly, the metadata persisted, and the asset is reachable from the public URL.

Recipe 544 The "Masquerade" Pattern

Lesson

Admin tools that switch users (masquerade, switch-user, sudo-to) need a test because they're exactly the kind of feature that breaks without anyone noticing.

Pattern from Varbase

```
@admin @local @development
Scenario: Admin masquerades as a normal user
  Given I am a logged in user with the "Webmaster" user
  And I am on "/admin/people"
  When I click "Masquerade" in the "Normal user" row
  Then I should see "You are now logged in as normal_user"
  When I am on "/user"
  Then I should see "normal_user"
  When I click "Unmasquerade"
  Then I should see "webmaster"
```

The pattern: switch in, verify, switch out, verify return.

Recipe

For any "switch user" admin tool, write a switch-and-return scenario. Test what happens if the user logs out mid-masquerade (the parent session should resume or end cleanly). Test that masquerade-as cannot bypass MFA on the target user.

Recipe 545 The Cache-Invalidation Test

Lesson

“Cache invalidation is one of two hard things in computer science.” It’s also one of two regression hotspots.

Pattern from Varbase

```
@cache @local @development
Scenario: Updating an article invalidates the listing page
  Given I am a logged in user with the "Content editor" user
  When I am on "/blog"
  Then I should see "Old Title"
  When I am on "/node/42/edit"
  And I fill in the field "Title" with "New Title"
  And I press "Save and keep published"
  When I am on "/blog"
  Then I should see "New Title"
  And I should not see "Old Title"
```

Recipe

Identify the listing pages that should refresh when a piece of content changes. Write a scenario: change the content, navigate to the listing, assert the new state. If the cache layer holds a stale value, this scenario fails.

Recipe 546 The Cross-Browser Smoke

Lesson

A test that passes in Chromium and fails in WebKit is a real bug. Catch it before users do.

Pattern from Varbase

The Varbase CI matrix runs the smoke suite across three browsers:

```
strategy:
  matrix:
    browser: [chromium, firefox, webkit]
```

With `BROWSER=${{ matrix.browser }}` plumbed through to `playwright.config.ts`.

Recipe

Run smoke-tagged scenarios across all three Playwright browsers. Run the full suite on chromium only, it's the cheapest and catches most issues. Reserve cross-browser for smoke and any known-finicky area (rich-text, drag-and-drop, custom video controls).

Recipe 547 The Update-Path Test

Lesson

A Drupal update path can break in ways unit tests can't detect. The only protection is to actually run the update and see what survives.

Pattern from Varbase

```
@update-path @local
Scenario: Update from 10.0 to 10.1 preserves content
  Given I am running Varbase 10.0
  And there are 5 published articles
  When I update to Varbase 10.1
  And I am on "/blog"
  Then I should see 5 published articles
  And I should not see "There was an error executing the update"
```

The first two `Given` steps are custom, they shell out to install a known version, then load fixture content.

Recipe

For any product with an update path (CMS, framework, library), build a regression scenario for each major version transition. The cost is low; the value is “we caught the migration bug before customers ran it on Friday at 5pm.”

Recipe 548 The Performance-Regression Sentinel

Lesson

A test that passes too slowly is also a kind of bug. Catch large regressions in performance.

Pattern from Varbase

```
@performance @smoke
Scenario: Homepage loads in under 2 seconds
  Given I start the performance timer
  When I am on "/"
  And I stop the performance timer
  Then the elapsed time should be under 2000 milliseconds
```

The performance steps are custom; the implementation is straightforward (record `performance.now()` at start and end).

Recipe

Add a small set of performance-sentinel scenarios. They shouldn't replace real performance testing, but they catch the 10x regression caused by a debug logger left on in production.

Recipe 549 The "No Broken Images" Sweep

Lesson

A page with an `<img>` whose `src` returns 404 is broken from the user's perspective. The page still renders, so no other test catches it.

Pattern from Varbase

```
Then(/^there should be no broken images$/, async function() {
  const broken = await this.page.evaluate(() => {
    return [...document.images]
      .filter(img => !img.complete || img.naturalWidth === 0)
      .map(img => img.src);
  });
  if (broken.length) {
    throw friendly(`Broken images:\n${broken.join('\n')}`);
  }
});
```

```
@smoke
Scenario: Homepage has no broken images
  Given I am on "/"
  Then there should be no broken images
```

Recipe

Add the “no broken images” assertion to every smoke scenario. The 10 lines of custom-step code catch one of the most common visible bugs in any visual CMS.

Recipe 550 The Pre-Release Suite

Lesson

Before a release, you want one fast, comprehensive sweep that confirms the basics. Build it as a tagged collection.

Pattern from Varbase

```
# Pre-release check
cucumber-js --tags "@smoke and @local"
```

Runs in 5 minutes. Covers the smoke tests, the basic permission matrix, and the most-trafficked workflows. If green, the release is safe to cut.

Recipe

Define what “safe to release” means in your project and tag the scenarios that prove it. Have engineering, QA, and product agree on the set. Run it before every release. When it goes red, halt.

Recipe 551 The “Tests Run In Parallel” Lesson

Lesson

A test suite that runs scenarios serially scales as O(N). One in parallel scales as O(N/cores). Big difference at 500 scenarios.

Pattern from Varbase

The Varbase CI matrix shards scenarios across 4 parallel jobs by feature directory. Each job runs ~125 scenarios in 7-9 minutes. Total wall time: 9 minutes for the full suite vs 36 minutes serial.

Recipe

When your suite passes 50 scenarios, look at parallelisation. Cucumber-js’s `--parallel N` flag is the cheapest entry point. CI-level parallelism (matrix sharding) scales further but needs careful test-data isolation.

Recipe 552 The “One Scenario, One Idea” Rule

Lesson

A scenario that tests three things will, on failure, give you three guesses about which thing broke. A scenario that tests one thing will tell you exactly.

Pattern from Varbase

Every Varbase scenario answers one question. The scenario name reads as the question:

```
Scenario: User can request a password reset
Scenario: Password reset email contains a working link
Scenario: Reset link expires after 24 hours
```

Three scenarios, three behaviours. The previous form was:

```
Scenario: Password reset works end-to-end   ← too big
```

Three failures merged into one ambiguous one.

Recipe

When the scenario name doesn’t fit on one line, split. When the scenario has more than ten steps, ask whether two scenarios would be clearer.

Recipe 553 The DAMP Principle for Tests

Lesson

Production code is DRY (Don't Repeat Yourself). Test code is DAMP (Descriptive And Meaningful Phrases). The two have different rules.

Pattern from Varbase

Varbase tests repeat short fragments deliberately:

```
And I press "Save"
Then I should see "The changes have been saved"
```

This duplicates across scenarios. We could DRY it into `When I save the form`. We don't, because the explicit version reads better in feature reports.

Recipe

Repeat short, readable phrases across scenarios. Extract only when the extraction adds genuine clarity (`I am a logged in user with the "X" user`). The cost of repetition is two extra lines per scenario; the benefit is readability for non-engineers.

Recipe 554 Test Data as Code

Lesson

Test data living in JSON files is fine until you need to derive one record from another, or compute a timestamp, or branch by environment. Then you want code.

Pattern from Varbase

```
// fixtures/articles.js
module.exports = {
  publishedArticle: (overrides = {}) => ({
    title: `Article ${Date.now()}`,
    status: 1,
    body: 'Lorem ipsum',
    ...overrides,
  }),
};
```

In a step:

```
const article = articleFixtures.publishedArticle({ title: 'My Article' });
```

The factory function makes derivation cheap.

Recipe

When fixture JSON starts repeating itself or needs computed values, move to factory functions. The pattern scales further than you'd expect, many large suites end up with a single `factories.js` per domain object.

Recipe 555 The Lessons-Learned Convention

Lesson

Every bug you fix is a lesson. Capture it as a test before you fix the bug.

Pattern from Varbase

Varbase's `tests/features/regressions/` directory has a feature per closed bug:

```
regressions/
  bug-1284-empty-coupon.feature
  bug-1456-stale-cache-after-publish.feature
  bug-1502-permission-cache-leak.feature
```

Each feature has one scenario: the smallest possible reproduction of the bug, plus the expected behaviour after fix.

Recipe

Adopt the convention. The next time someone files a bug, write the reproduction scenario before writing the fix. The test goes red. You fix the bug. The test goes green. The bug never returns, which is exactly what regression suites are for.

Recipe 556 The Webship-js Origin Story

Lesson

We did not set out to build webship-js. We set out to test Varbase. Webship-js is what we extracted from the Varbase test rig once we noticed we were rebuilding the same thing every time a new project started.

The pattern recurred:

1. New project starts. Cucumber + Behat + Selenium gets stood up.
2. Step definitions are written, copy-pasted from the last project.
3. After three months, the step definitions diverge silently.
4. Bugs in one project's step definitions don't get fixed in the other.

The fix was to make the step definitions a package: install once, share across projects. Webship-js is that package.

Recipe

If you find yourself rebuilding the same test rig across projects, extract it. The cost of extraction is two days. The cost of not extracting is a decade of divergence.

Recipe 557 The Drupal-First, Framework-Agnostic Path

Lesson

Webship-js was born from a Drupal test suite. Its first hundred features were Drupal-specific. But every step we wrote was about user behaviour, not about Drupal, “log in”, “fill in a field”, “save a node”. When we generalised to non-Drupal projects, almost everything carried over.

Pattern

Look at the step names. None of them mention Drupal. The Drupal-ness lives in step definitions and fixtures, not in features.

Recipe

When you write a step, ask: does the step name describe user behaviour or framework behaviour? If it’s framework behaviour, hide it behind a user-behaviour name. Your feature files become portable across stacks.

Recipe 558 The Vardot Stack as Reference Implementation

Lesson

We use webship-js on every Vardot project. The Vardot stack is, in effect, the reference implementation, the codebase we hit every edge case against.

If you want to see webship-js in production, look at:

- The Varbase 10.1.x test suite (Drupal 10)
- The Varbase 11.0.x project test suite (Drupal 11)
- Vardot client projects (private, but the patterns are documented here)

Recipe

When you adopt webship-js, treat the Varbase test suite as your reading room. The patterns we ship in this book are the ones we'd adopt in any new project, distilled from years of running Varbase tests. Adapt, don't copy verbatim, but the shape will help you start.

Recipe 559 What We Got Wrong (And Then Fixed)

Lesson

Three things we got wrong in the early days:

1. **Selenium-style waits everywhere.** Every step had a manual `wait for AJAX` after it. Tests were slow and brittle. We added web-first assertions in webship-js ~2.0; now `wait for AJAX` is rare in new tests.
2. **JSON fixtures, not factories.** Test data was 30 huge JSON files that became impossible to keep in sync. Moving to factory functions cut maintenance to a fraction.
3. **One giant feature per area.** A 60-scenario feature was unreadable. Splitting by behaviour (one feature per workflow, not one per area) made reviews 5x faster.

Recipe

If your suite resembles any of those three early-Varbase patterns, plan a migration. The cost is real but bounded; the payoff is permanent.

Recipe 560 The Long Game

Lesson

A test suite is a long-term asset. The patterns that matter most aren't the clever ones; they're the boring ones. Naming conventions. Tag taxonomies. Stable fixtures. Predictable file structure. Small scenarios. Readable failures.

Pattern from Varbase

Ten years in, the suite that started as one developer's Behat experiment has run on every CI run for thousands of releases. The patterns that survived are in this chapter. The ones that didn't, single-file mega-features, mocked DBs, complex inheritance hierarchies in step definitions, got refactored out years ago.

Recipe

Build for the version of yourself who maintains the suite three years from now. They will not remember the clever pattern. They will, however, appreciate the boring one that still works.

We learned a lot by experimenting and working on Varbase. Every recipe in this book carries some fingerprint of that experience, the patterns kept, the patterns discarded, and the surprises that taught us how acceptance testing actually works in production. If you only adopt one chapter into your own practice, make it this one.

Chapter 48: Storage, Cookies, and Selectors, Deep Dive

Step definitions and code examples by Tasneem Natshah, from the webship-js source.

Three subsystems that sit underneath the rest of the test suite, the **cookie jar**, the **storage trio** (localStorage, sessionStorage, IndexedDB cleanup), and the **named-selector registry**. Webship-js 2.0 expanded all three into rich families of steps. This is the deep-dive chapter that covers every one, with practical recipes.

Here is why these three deserve a chapter of their own: when a suite that has been green for months starts failing in ways nobody can explain, the root cause is almost never the assertion that failed. It is a cookie that leaked from a previous scenario, a localStorage key the app wrote and nobody cleared, or a selector that three feature files each define slightly differently. If you've worked through Chapter 10 (authentication) and Chapter 34 (advanced selectors), this chapter goes one layer deeper: variant cookie matchers, storage cleanup as a hook pattern, named selectors loaded from JSON, runtime selector additions, and the layout assertions that build on all of it.

Cookie Family Overview

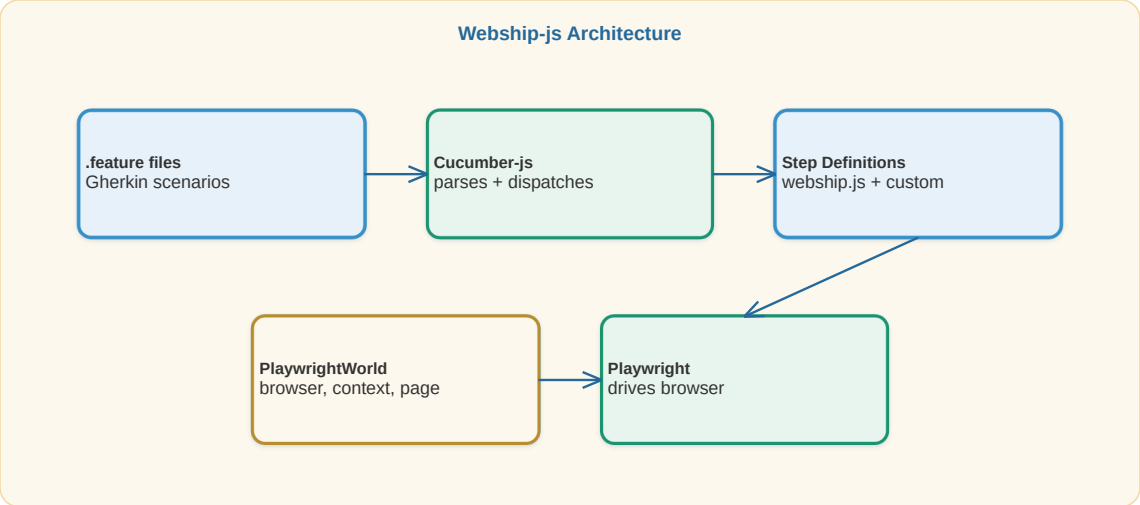


FIGURE 48.1 *Cookie Family Overview*

Cookies have **four orthogonal questions** you might ask about them:

1. Is the name exact, or substring? (`name "session"` vs `name containing "sess"`)
2. Is the value exact, or substring? (`value "abc123"` vs `value containing "abc"`)
3. Should it exist, or not? (`should exist` vs `should not exist`)
4. Are we asserting, or setting?

Webship-js ships dedicated steps for every combination. Four name-and-value combinations, each with a `should exist` and a `should not exist` form, gives eight. Add the name-only checks (exact and containing, positive and negated) and you land on **twelve cookie matchers**, plus three setters (`is set to`, `is removed`, `all cookies are cleared`), enough to cover any cookie-related test you'd write.

All twelve matchers read the jar the same way: through Playwright's `context.cookies()` API, at the browser-context level. That one design decision matters more than it looks. `context.cookies()` sees **every** cookie in the context, including `HttpOnly` cookies that page

JavaScript and `document.cookie` can never read. Since real session cookies are almost always `HttpOnly`, a cookie assertion that ran through `document.cookie` would be blind to exactly the cookies you care about most. These steps are not.

Recipe 566 Setting a Cookie

Problem

Your site reads `theme=dark` from a cookie and renders the dark layout server-side. The only way to create that cookie through the UI is a four-click trip through the preferences page, which means every dark-mode test spends its first ten seconds testing the preferences page instead of the thing it's actually about. You want the cookie to simply exist before the page loads.

Solution

```
Feature: Theme cookie

  Scenario: Dark theme renders when cookie is set
    Given the cookie "theme" is set to "dark"
    When I am on "/"
    Then "body" should have class "theme-dark" within 1 second
```

The cookie is set before navigation. The page sees it on first render.

Under the hood the step calls Playwright's `context.addCookies()`. It needs a domain to attach the cookie to, so it looks at the current page URL, and if the page is still on `about:blank` (which it is, at the start of a scenario), it falls back to the hostname of your `launchUrl`. Path defaults to `/`. Because the cookie lives at the browser-context level, not the page level, it is already in the jar when the first request goes out, the server receives it in the `Cookie` header of the very first HTML request, which is why server-rendered dark mode works on first paint. The closing assertion is a web-first matcher: it polls the page every 100ms for up to one second, so a class applied by a late-running script still passes.

Pro Tip: This pre-navigation trick works for cookies because they are context-scoped. It does *not* work for localStorage, which belongs to an origin the browser hasn't visited yet, see Recipe 574 for the storage equivalent.

Recipe 567 Removing a Cookie

Problem

A user's session expires while they have a tab open. They click something, and instead of a polite redirect to the login page they get a white screen and a stack trace, you know, because that exact ticket landed on your desk last quarter. To test the fix you need a logged-in browser whose session cookie suddenly vanishes, mid-scenario, while everything else stays intact.

Solution

```
Scenario: Expired session redirects to login
  Given I restore the auth state from "tests/auth/webmaster.json"
  And I am on "/admin/content"
  When the cookie "session" is removed
  And I reload the page
  Then the path should be "/user/login"
```

The cookie is removed without clearing the rest of the cookie jar, preserves test data set in earlier steps.

Two pieces of machinery are doing work here. The `restore the auth state` step (from `auth.steps.js`) opens a fresh browser context seeded from a saved `storageState` JSON snapshot, cookies plus localStorage from a real login that happened once, earlier, and re-installs the webship-js init script so smart waits keep working. That's how the scenario is "logged in" without walking through the login form. The removal step then reads the full jar via `context.cookies()`, filters out the one cookie by name, clears everything, and re-adds the sur-

vivors. The result is surgical: language preference, consent flag, A/B bucket all stay; only the session dies. The reload sends the next request without the session cookie, and the server does whatever your fix says it should do.

At scale: the snapshot files are the part that pays off across projects. Keep one per role per project, `tests/auth/admin.json`, `tests/auth/editor.json`, generated by a one-shot `@auth-setup` scenario you re-run after every credential rotation. Twenty projects with the same layout means anyone on the team can open any suite and know exactly where authentication comes from, and your CI secrets live in one setup scenario instead of being sprinkled through every feature file.

Recipe 568 Clearing All Cookies

Problem

The cookie-banner test passes when you run the feature file alone and fails when the whole suite runs, because some earlier scenario accepted the banner, and your "first visit" is no longer a first visit. It's the classic Tuesday-before-release discovery: the test was never wrong, the starting state was.

Solution

```
Scenario: Anonymous visit
  Given all cookies are cleared
  When I am on "/"
  Then ".cookie-banner" should be visible within 2 seconds
```

The clear is total, session, preferences, consent flags, everything. Use sparingly; most tests benefit from per-scenario isolation via the Cucumber lifecycle, not manual clears.

The step is one line of Playwright: `context.clearCookies()`. Worth knowing before you reach for it: webship-js already opens a **fresh browser context for every scenario** in its `Before` hook, and a fresh context has an empty jar. So in a standard suite the example above is belt-and-braces, the real use cases are mid-scenario resets (log in, do something, then verify the anonymous view of

the same page) and scenarios that restored an auth snapshot and now need to shed it. There is also a heavier sibling: `Given I am an anonymous user` clears cookies *and* navigates to your `launchUrl` in one step, which reads better when "fresh visitor on the homepage" is the actual precondition.

At scale: if a scenario only passes after a manual cookie clear, treat that as a smell, not a fix. Across 20 projects, suites that depend on scenario *order* are the ones that go green locally and red in CI when parallel workers shuffle execution. The per-scenario fresh context is your isolation guarantee, write every scenario as if it runs first, and reserve explicit clears for mid-scenario state changes you can point to in the feature text.

Recipe 569 Asserting Cookie Exists by Exact Name

Problem

A login that *looks* successful, right redirect, right welcome message, but never set the session cookie will bite you three pages later, when the user is mysteriously anonymous again. The new hire who debugs this opens DevTools, types `document.cookie`, sees nothing, and loses an afternoon, because session cookies are `HttpOnly` and invisible to page JavaScript. You want the test to check the jar directly, at the level where the cookie actually lives.

Solution

```
Scenario: Login sets the session cookie
  Given I am on "/user/login"
  When I fill in "username" with "normal_user"
  And I fill in "password" with "dD.123123ddd"
  And I press "Log in"
  Then a cookie with the name "SESS_a1b2c3" should exist
```

`should exist` is the strict, name-based existence check.

The step fetches the live jar with `context.cookies()` and does an exact `name ===` comparison. Because it reads at the browser-context level, it sees `HttpOnly` and `Secure` cookies that `document.cookie` cannot, which is precisely what makes it useful for session cookies. On failure you get `Cookie "SESS_a1b2c3" was not found.` rather than a downstream symptom three steps later. One thing this step deliberately does *not* do is wait: it reads the jar at the moment it runs. That's fine here because `I press "Log in"` is a state-changing step, and webship-js runs `smartSettle` after it, the response (and its `Set-Cookie` header) has landed before the assertion fires.

Pro Tip: the hash suffix in `SESS_a1b2c3` is environment-specific, this exact-name assertion will pass on dev and fail on staging. If your suite runs against more than one environment, you want the next recipe.

Recipe 570 Asserting Cookie by Partial Name

Problem

Drupal's session cookie name includes a hash that changes per environment, `SESS_a1b2c3` on dev, `SESS_x9y8z7` on staging. The login test you wrote against dev goes red the first time the pipeline points it at staging, and someone "fixes" it by copying the feature file and hard-coding the other hash. Now you maintain two tests that say the same thing. You want one test that works everywhere.

Solution

```
Scenario: Login sets a session cookie
  Given I am on "/user/login"
  When I fill in "username" with "normal_user"
  And I fill in "password" with "dD.123123ddd"
  And I press "Log in"
  Then a cookie with a name containing "SESS" should exist
```

`a name containing` does substring matching. Useful for cookies whose names are environment-dependent or library-generated.

Implementation-wise it's the same `context.cookies()` read as the exact matcher, but the comparison becomes `name.indexOf(partial) !== -1`, a plain substring check, scanned across every cookie in the jar until one matches. That covers Drupal's per-site hash, framework-generated names (`csrftoken_`, `XSRF-`), and analytics cookies with versioned suffixes. The trade-off is honesty about looseness: `containing "SESS"` would also match a hypothetical `SESSION_DEBUG` cookie. Pick the longest stable fragment you have.

At scale: environment-portable assertions are what let one feature file serve your whole pipeline. The pattern that works across many projects: the suite takes its target from the `LAUNCH_URL` env var, and every assertion inside the features is written to pass on *any* environment, partial cookie names, path assertions instead of full URLs, label-based field lookups. The moment a feature file contains an environment-specific literal, you've forked your suite without noticing.

Recipe 571 Asserting Cookie Value

Problem

The consent banner sets `consent=accepted` when the user clicks "Accept all", and the legal team has made it very clear that "the cookie exists" is not the same claim as "the cookie says accepted". A refactor that accidentally writes `consent=1` instead would pass an existence check and still put you out of compliance. You need the value pinned, exactly.

Solution

```
  Scenario: Consent cookie value
    Given I am on "/"
    When I click "Accept all"
    Then a cookie with the name "consent" and the value "accepted" should
exist
```

Both name and value match exactly.

The click step finds a link or button whose visible text is exactly “Accept all”, under the hood it filters `a, button, [role="button"], input[type="submit"]` by text, so it works whether the banner ships a `<button>` or a styled link. The assertion then finds the cookie by exact name and compares the value with strict equality. When the value is wrong you get the actual vs. expected in the failure message, `Cookie "consent" has value "1", expected "accepted".`, which turns a compliance bug from an investigation into a one-line diff. This is the strictest of the twelve matchers; use it whenever the value is a contract, not an implementation detail.

Recipe 572 Asserting Cookie Value by Substring

Problem

A tracking cookie stores a JSON blob: `{"opt_in":true,"id":"abc"}`. The `id` changes every session, so an exact-value assertion fails on every run, and the first instinct, asserting the whole JSON with a wildcard id, couples your test to field order and every future key the analytics team adds. You care about one fact: `opt_in` is true. Assert that slice and nothing else.

Solution

```
  Scenario: Tracking opt-in is set
    Given I am on "/"
    When I click "Allow tracking"
    Then a cookie with the name "tracking" and a value containing
"opt_in\":true" should exist
```

Substring matching on the value, great for JSON payloads where the test only cares about a slice.

The matcher finds the cookie by exact name, then runs `value.indexOf(partial) !== -1` against the raw stored value. Note the `\"` in the Gherkin, Cucumber’s `{string}` parameter supports backslash-escaped quotes, so the actual substring searched for is `opt_in":true`, ex-

actly as it sits inside the JSON. The recipe survives new keys, reordered fields, and rotating ids, because none of those touch the five characters you pinned.

Pro Tip: `context.cookies()` returns the value exactly as stored, and many frameworks URL-encode cookie values, so your JSON may actually live in the jar as `%7B%22opt_in%22%3Atrue...%7D`. If the substring assertion fails against a cookie you can see in DevTools, encoding is the first suspect. Check the raw value once (a temporary custom step that prints `await this.context.cookies()` does the job), then match a fragment that survives the encoding your app uses.

Recipe 573 The Four-Way Cookie Matrix

Teams that test cookies across several projects keep reinventing this table, usually badly, in a Slack thread, the day someone needs the negated-substring form and can't remember whether it exists. Here is the full matrix in one place:

NAME	VALUE	STEP
exact	exact	`a cookie with the name "X" and the value "Y" should exist`
exact	contains	`a cookie with the name "X" and a value containing "Y" should exist`
contains	exact	`a cookie with a name containing "X" and the value "Y" should exist`
contains	contains	`a cookie with a name containing "X" and a value containing "Y" should exist`

All four have `should not exist` negations, that's eight. Add the name-only pair from Recipes 569-570 (`a cookie with the name "X" should exist` / `a name containing "X" should exist`) and their negations, and you reach the full twelve. Every one of them is the same two-line

pattern inside: read `context.cookies()`, then match with `===` or `indexOf` depending on the row you picked.

Use the most specific form your test can stand. Looser matches hide more bugs: `a name containing "consent"` happily passes while the real cookie is `consent_v2_legacy_do_not_use`. The negated forms deserve special respect, `a cookie with the name "tracking" should not exist` *after declining consent* is one of the few assertions that catches a privacy regression before your users (or a regulator) do.

At scale: put this table in your team's testing conventions doc and reference it in review. When 10 suites across 10 projects each assert cookies in their own dialect, some through custom steps, some through `document.cookie` evaluations, review can't tell a deliberate loose match from a lazy one. A shared rule like "exact name and value for contract cookies, substring only for generated names, negation required for consent declines" makes every cookie assertion reviewable by anyone on the team, not just the person who wrote it.

Recipe 574 LocalStorage Setting and Removal

Problem

Your site shows a "Continue reading" link pointing at whatever the visitor last read, the app keeps a `last_page` key in localStorage. Testing it through the UI means three navigations: open the article, leave, come back. And the first person who copies the cookie recipe from earlier in this chapter, seed the value, *then* navigate, gets a failure that says `Access is denied for this document` and loses an hour to it. Storage is not cookies. It needs a page first.

Solution

```
  Scenario: Last-viewed page is restored
    Given I am on "/"
    And the local storage "last_page" is set to "/articles/42"
    When I reload the page
    Then the link "Continue reading" with the href "/articles/42" should
exist
```

Removal is the same shape, one key gone, every other key untouched:

```
  Given the local storage "last_page" is removed
```

The set step is a single `page.evaluate()` call running `localStorage.setItem(key, value)` on whatever page is currently open. That one fact explains the step order, and it is the exact mirror of Recipe 566's Pro Tip. Cookies live at the browser-context level, so you can plant them before any navigation. localStorage belongs to an *origin*, and at the start of a scenario the page is still parked on `about:blank`, which has no origin storage at all; touching `localStorage` there throws. So the working sequence is always: navigate (now the origin exists), seed the key, reload (the app boots again and reads what you planted). The reload is not optional, the app finished its startup before you wrote the key, so it has to start over to notice. One more thing the step will not do for you: stored values are always strings. JSON-encode objects yourself, Recipe 594 shows that in anger.

Recipe 575 Clearing All LocalStorage

Problem

You are testing the first-visit welcome tour, but by the time your scenario reaches the interesting part, the app has already written `tour_seen=true`, your own earlier steps triggered it. You need to wipe storage *mid-scenario* and watch the page behave like it has never met this visitor. The in-

stinct is to clear before navigating, the way the cookie recipes do, and that is exactly the order that fails.

Solution

```
Scenario: Fresh visit, empty storage
  Given I am on "/"
  And local storage is cleared
  When I reload the page
```

Equivalent for session storage:

```
Given session storage is cleared
```

The step runs `localStorage.clear()` inside the page, per-origin, same `about:blank` rule as Recipe 574, which is why the navigation comes first. Two scoping facts keep this honest. First, "all" means all *for the current origin*: `app.example.com` and `admin.example.com` hold separate storage, and clearing one says nothing about the other, cookies, by contrast, can span subdomains. Second, you need this step less often than you think. Webship-js opens a fresh browser context for every scenario, and a fresh context starts with empty storage everywhere, so the clear at the top of a scenario is usually redundant. Reach for it for mid-scenario resets like the tour example, the same advice Recipe 568 gives for cookies: if a scenario needs a clear at the top just to pass, something else is leaking, and the clear is hiding it.

Recipe 576 Session Storage Set and Remove

Problem

Your checkout wizard keeps its progress in sessionStorage under `wizard_step`, so a refresh keeps the user where they were but a fresh tab starts over. QA needs to land on step 3 without clicking through steps 1 and 2 every single run, and last sprint someone wrote that walk-through as

a fixture, then watched it break every time the step-2 form changed. Seed the key, open the page, and the wizard resumes where you told it to.

Solution

```
Scenario: Wizard resumes at step 3
  Given the session storage "wizard_step" is set to "3"
  When I am on "/wizard"
  Then ".wizard-step-3" should be visible within 2 seconds
```

Session storage is per-tab, it clears on tab close. Use it for ephemeral state that shouldn't survive a restart.

The set step is a `page.evaluate()` running `sessionStorage.setItem(key, value)`, the sessionStorage twin of Recipe 574's `localStorage.setItem`, and it lives under the same origin rules. The example reads as if it sets the key *before* navigating, but read the order again: the `Given` runs against whatever page is open, and at scenario start that is `about:blank`, which has no origin and throws on any storage access. In practice you navigate first, seed the key, then trigger the read, the same navigate-seed-reload dance Recipe 574 spells out. The removal form is `Given the session storage "wizard_step" is removed`, and like its localStorage sibling it deletes one key and leaves the rest of the tab's storage alone. The values are strings; JSON-encode anything richer yourself.

At scale: sessionStorage and localStorage look interchangeable until you run them across a fleet, and then the difference bites. sessionStorage dies with the tab, so a scenario that seeds it survives a `reload` but not a `start/stop video recording` step, that rebuilds the browser context and takes the tab (and its session storage) with it (see the video chapter). When you template wizard tests across 20 product suites, write a one-line comment in the feature noting which storage the app actually uses; the new hire who copies your scenario into a localStorage-backed wizard otherwise spends an afternoon wondering why a seeded value vanished on the next tab.

Recipe 577 Storage Hook Pattern

Problem

A suite that has been green for a month starts failing one scenario in three, and only in CI, never locally. The cause is always the same shape, one scenario writes a localStorage key, the next scenario assumes a clean slate, and which one runs first depends on how the parallel workers shuffled the order that night. You want a single place that guarantees every scenario starts with nothing in cookies or storage, so order stops mattering.

Solution

```
const { Before } = require('@cucumber/cucumber');

Before(async function() {
  await this.page.context().clearCookies();
  await this.page.evaluate(() => {
    localStorage.clear();
    sessionStorage.clear();
  });
});
```

Drop into your global hooks file. Every scenario starts truly clean.

Here is the thing worth knowing before you add this: webship-js already opens a **fresh browser context for every scenario** in its own `Before` hook, and a fresh context has an empty cookie jar and empty storage everywhere. So `clearCookies()` here is belt-and-braces. The `localStorage.clear()` and `sessionStorage.clear()` calls run inside the page via `page.evaluate()`, and they are per-origin, they wipe storage for the origin the page is currently sitting on, nothing more. At the very top of a scenario the page is still on `about:blank`, which has no origin storage and throws when you touch `localStorage`; this hook survives only because it runs after the framework's own setup has navigated somewhere with an origin. If you ever

see it blow up with `Access is denied for this document`, that is the `about:blank` problem, Recipe 591 wraps each clear in a `try/catch` precisely so a no-origin page degrades quietly instead of failing the run.

At scale: this hook is the single most copied snippet across a multi-project test estate, which is exactly why it should live in one shared hooks file you publish once, not re-typed into 20 repos that then drift. When every suite isolates state the same way, a red scenario in CI means a real regression, not a leaked cookie, and that is the difference between a PMO dashboard people trust and one they have learned to ignore. The corollary is a review rule: a scenario that only passes because *another* scenario left state behind is a bug in the test, and this hook is what makes that bug impossible to write by accident.

Recipe 578 Defining a Single Named Selector Inline

Problem

The login button is buried behind `nav.primary a[data-role='auth-cta']:not(.disabled)`, and that exact string is now pasted into nine scenarios. The day the design team renames `data-role`, you are doing find-and-replace across nine files and praying you caught them all. You want one short name, `login-button`, that every step can use, defined once.

Solution

```
  Scenario: Login uses named selector
    Given I add "login-button" selector for "nav.primary a[data-
role='auth-cta']:not(.disabled)" css selector
    When I am on "/"
    And I click "login-button"
    Then ".user-menu" should be visible within 2 seconds
```

`add ... css selector` registers the name in the per-scenario registry. Subsequent steps can refer to `"login-button"` instead of the full CSS.

The registry is part of the world's `selectors` config, a plain object shaped `{css: {}, xpath: {}}` that lives for the life of one scenario. This step writes `"login-button"` into the `css` map. From then on, every step that takes a selector tries the registry first: when `I click "login-button"` runs, the resolver finds the name, swaps in the full CSS, and hands that to Playwright's `getByRole`/locator machinery. A name that *isn't* in the registry falls straight through and is treated as a literal selector, which is why `.user-menu` on the next line just works without being registered. The registration is per-scenario by design, the next scenario starts with whatever the world bootstraps (the files in `worldParameters.selectors.files`) and none of your inline additions, so one scenario can never pollute another's vocabulary.

At scale: inline `add ... selector` is the right tool for a selector that genuinely belongs to *one* scenario. The moment a name shows up in a second scenario, it has earned a home in a shared JSON file (Recipe 580) instead. The pattern that holds up across many projects is a layered vocabulary: a shared cross-project file for the handful of selectors every site has (`main-nav`, `search-box`), a per-project file for that product's quirks, and inline additions only for the genuinely one-off. Review can then ask one question, "why is this selector inline instead of in the file?", and usually the answer reveals either a one-off (fine) or a missed chance to share (fix it).

Recipe 579 Defining Multiple Selectors in Bulk

Problem

The complex-form scenario needs ten named selectors, and ten separate `add ... selector` lines push the actual test, the three steps that matter, below the fold of anyone's editor. A reviewer scrolling past ten setup lines to find one assertion stops reading carefully, which is exactly when bugs slip through. You want the whole vocabulary declared in one readable block, then the test.

Solution

```
  Scenario: Complex form uses many selectors
    Given I define css selectors:
      | name           | selector
|
      | login-button   | nav.primary a[data-role='auth-
cta']:not(.disabled) |
      | name-field     | input[name='username']
|
      | password-field | input[name='password']
|
      | submit-button  | button[type='submit'].btn-primary
|
    When I am on "/user/login"
    And I fill in "name-field" with "user"
    And I fill in "password-field" with "pass"
    And I click "submit-button"
```

One step, a data table, every selector registered. Use this when a scenario has five or more named selectors.

`I define css selectors:` takes a Cucumber data table, two columns, `name` and `selector`, and loops every row into the same per-scenario `css` registry that Recipe 578 writes to one entry at a time. There is no behavioural difference between this and four `add` steps; it is the same map ending up with the same keys. What changes is readability, and that is the whole point: the resolver still tries the registry first for every selector-taking step, so `name-field` resolves through `input[name='username']` to label-first field resolution, and `submit-button` reaches `getByRole` through `button[type='submit'].btn-primary`. The xpath twin is `I define xpath selectors:` (Recipe 582), writing to the separate `xpath` map.

At scale: a bulk-define block at the top of a scenario is a sign you are about to share these selectors. Treat the table as a staging area, the day a second scenario needs the same four names, lift the table into a JSON file (Recipe 580) and let `worldParameters.selectors.files` auto-load it. Across 20 suites the rule of thumb is simple: inline single adds for one-offs, a define-table when a

single scenario needs many that are still genuinely local, and a JSON file the instant anything is reused. The table format also diffs cleanly in code review, so a selector change reads as one row changed, not a buried edit inside a long Gherkin line.

Recipe 580 Loading Selectors From a JSON File

Problem

The same `login-button` selector is defined inline in a dozen scenarios across four feature files, and they have already drifted, two of them still point at the old `data-role`, and nobody noticed until a flaky login test turned out to be two different selectors wearing the same name. You want the selector defined in exactly one place, read by every scenario automatically, so “the login button” means one thing in the whole suite.

Solution

Create `tests/selectors/auth.json`:

```
{
  "css": {
    "login-button": "nav.primary a[data-role='auth-cta']:not(.disabled)",
    "name-field":   "input[name='username']",
    "password-field": "input[name='password']"
  }
}
```

Reference it in `worldParameters.selectors.files`:

```
// cucumber.js
module.exports = {
  default: {
    worldParameters: {
      selectors: { files: ['tests/selectors/auth.json'] }
    }
  }
};
```

Now every scenario inherits the file's selectors. Override per-scenario with inline `add ... selector` steps when needed.

This is the registry's intended home. Each file is the same `{"css": {}, "xpath": {}}` shape as the in-memory registry, which is no accident, at world setup webship-js reads every path in `worldParameters.selectors.files`, resolves it against `selectors.filesPath` (default `'./tests/selectors/'`), and merges the file's `css` and `xpath` maps into the scenario's starting registry. So by the time your first step runs, `login-button` is already a known name in every scenario, with zero Gherkin spent registering it. Inline `add ... selector` steps still work and layer on top of the file-loaded names, which is what "override per-scenario" means: a later `add` for an existing name wins for that scenario only, and the next scenario reverts to the file's version. Keep the file rooted under `tests/selectors/` and `files: ['auth.json']` resolves just as well as the full path.

At scale: the JSON file is the artifact that makes a selector vocabulary shareable across an entire estate, and it is plain JSON on purpose, a non-developer can read it, a PO can confirm "login-button" points where they expect, and DevOps can swap it per environment without touching a single feature file. The pattern that holds across 20 projects: a shared base file (published as an npm package, Recipe 592) for the selectors every site has, plus a thin per-project file for that product's quirks, both listed in `files:`. A selector rename then becomes a one-line edit in one file that the whole suite picks up on the next run, the opposite of the drift you started with.

Recipe 581 Adding Selectors From a File Mid-Scenario

Problem

Your admin-area scenarios need 30 admin-only selectors, but the other 200 scenarios in the suite never touch the admin UI. Dumping all 30 into the global `files:` list means every front-end test loads selectors it will never use, and the names start colliding with public-facing ones, a `toolbar` in admin is not the `toolbar` on the marketing page. You want those selectors loaded only for the scenarios that actually need them.

Solution

```
@admin
Scenario: Admin uses admin selectors
  Given I add selectors from "tests/selectors/admin.json" file
  When I am on "/admin"
  And I click "admin-toolbar-cta"
```

The file is loaded into the scenario's registry. The next scenario starts without it, registry is per-scenario.

This step does the same file read as the global `files:` loader (Recipe 580), parse the JSON, merge its `css` and `xpath` maps into the registry, but at step time, into one scenario's registry only. Because the registry is rebuilt fresh for every scenario from `worldParameters.selectors.files`, anything you load mid-scenario evaporates when that scenario ends; the next one starts with the global baseline and nothing else. That is the scoping you want for an admin file: the 30 admin names exist exactly where the `@admin` tag says they should, and a `toolbar` selector in `admin.json` can shadow a global `toolbar` for that scenario without leaking into the front-end tests that ran before or after.

At scale: the global-versus-mid-scenario choice is really a namespace-collision strategy across many projects. Selectors that every scenario uses belong in the global `files:` list; selectors scoped to one section, admin, a checkout flow, an embedded third-party widget, belong in their

own file loaded behind a matching tag. Pair the load step with the same tag your CI uses to slice the run (`@admin`), and DevOps gets a clean story: the admin selector file only loads when admin scenarios run, so a front-end-only nightly pass never parses it. The tag, the file, and the section it covers all carry the same name, which is the kind of consistency that lets someone open the twentieth project's suite and immediately know where the admin selectors come from.

Recipe 582 Defining XPath Selectors

Problem

You need to click the Edit link in the row whose cells contain "Article", and CSS has no way to say "the parent row that contains this text". The team's first attempt was an `:nth-child` index, which broke the moment the content list re-sorted. This is one of the handful of cases where XPath earns its keep, and you want to name the ugly expression so the scenario stays readable.

Solution

```
Scenario: XPath registers as a named selector
  Given I define xpath selectors:
    | name        | xpath                                                |
    | article-row | //tr[td[contains(.,'Article')]]                      |
    | edit-link   | //tr[td[contains(.,'Article')]]//a[text()='Edit']    |
  When I am on "/admin/content"
  And I click "edit-link"
```

XPath and CSS selectors live in separate registries but are searched together at resolve time. The CSS registry wins on name collision, prefer CSS for new selectors when both would work.

`I define xpath selectors:` is the structural twin of Recipe 579's CSS define-table, except every row lands in the world's `xpath` map instead of `css`. The two maps are deliberately separate, but the resolver checks both when a step hands it a name, so `edit-link` resolves the same way whether it was registered as CSS or XPath, and Playwright is told which engine to use based

on which map held the name. When the *same* name exists in both, CSS wins, which is why the guidance is to reach for XPath only when CSS genuinely can't express the relationship, ancestor/text predicates like the one above. JSON files carry XPath the same way: a `{"xpath": {...}}` block in any file from Recipe 580 auto-loads alongside its `css` sibling.

At scale: XPath is the selector style that ages worst, so the registry is also a containment strategy. Across many projects, structural XPath like `//tr[td[contains(.,'Article')]]` is the first thing to break when a CMS upgrade reshapes its markup, and when that expression is pasted inline in 15 scenarios, you re-fix it 15 times. Named in one JSON file, it is one edit. Make it a convention that XPath always goes in the file behind a name, never inline in a feature, with a one-word comment on *why* CSS wasn't enough; the next person upgrading the platform then has a single, documented list of the brittle selectors to re-verify, instead of hunting them across the whole suite.

Recipe 583 Printing the Active Selector Registry

Problem

A step fails with `Could not find "foo button"`, and you are certain you registered it. The usual culprits are invisible: a trailing space in the name, a file that didn't load because the path was wrong, or a JSON file that silently merged into the wrong map. Rather than guess, you want to see exactly what names the resolver actually has right now.

Solution

```
Scenario: Debug the registry
  Given I define css selectors:
    | name        | selector |
    | foo button  | .foo-btn |
  Then print css selectors
  And print xpath selectors
```

Both `print` steps dump the active registry to stdout. Use them during scenario authoring; remove before committing.

These two steps print the world's live `css` and `xpath` maps at the moment they run, the merged result of everything: the files from `worldParameters.selectors.files`, any mid-scenario file loads (Recipe 581), and every inline `add` / `define` up to that line. That is what makes them the right debugging tool. If `foo button` is missing from the printed CSS map, you registered it somewhere that didn't take, wrong file, wrong map, a typo in the name. If it is present but the step still fails, the name resolved fine and your CSS itself doesn't match anything on the page, which sends you to the DOM, not the registry. Because the maps print as plain key/value, a trailing space or a case mismatch jumps straight out at you.

Pro Tip: these are authoring-time steps, not assertions, they print and pass, they never fail a build. Pull them out before you commit, the same way you would a `console.log`. A `print css selectors` left in a feature file clutters every CI run's stdout and trains your team to ignore output that should mean something. If you find yourself needing them often in one project, that is usually a sign the selector file isn't loading the way you think, fix the `files:` path once rather than debugging it scenario by scenario.

Recipe 584 CSS Property Assertions

Problem

A theme refactor moved the brand colour into a CSS custom property, and three weeks later the primary call-to-action button quietly rendered grey on production because a variable name had a typo nobody caught. Visual review missed it; it was a subtle shade off. You want a test that pins the actual rendered colour of the button, so "the CTA is the brand blue" is a fact the suite checks, not a thing humans eyeball.

Solution

```
  Scenario: Hover changes button background
    Given I am on "/"
    When I hover over "#cta-button"
    Then "#cta-button" should contain cssproperty "background-color" with
value "rgb(53, 143, 200)"
```

CSS properties are read via `getComputedStyle()`, so the value is the resolved final colour, not the raw declaration. Match against the resolved form (`rgb(...)` not `#358FC8`).

The assertion reads the property off the element's *computed* style, which is the value the browser actually paints after cascade, custom properties, and media queries have all resolved. That is why you match `rgb(53, 143, 200)` and not `#358FC8` or `var(--brand)`, by the time `getComputedStyle()` sees it, every layer has collapsed to a concrete `rgb()` (or `rgba()` if there is any alpha). It also means this catches the broken-variable bug from the Problem directly: a mistyped `var(--brnd)` falls back to its default and the computed colour changes, so the assertion fails with the actual vs. expected colour. The `hover over` step moves the real pointer to the element first, so a `:hover`-only colour is genuinely in effect when the property is read.

At scale: colour and computed-style assertions are powerful and brittle in equal measure, so spend them where a value is a contract, not everywhere. Across many projects, hard-coded `rgb(...)` literals scattered through features are the assertions that go red en masse the day a design system re-tokenizes its palette, and then the failure is the test, not the product. The pattern that survives: assert computed style only on the handful of elements where the colour is the point (brand CTA, error state, the disabled affordance from the next recipe), and assert the rest through `should have class` so a token change flows through without touching a single feature. When you do pin a colour, name the expected value once and reference it, so a palette bump is one edit across the estate, not a sweep.

Recipe 585 Negated CSS Property Assertions

Problem

A form's Save button is supposed to look obviously dead until the form is valid, but a CSS regression left it rendering in full brand blue while still disabled, so users kept clicking a button that did nothing, and the support queue filled up. The trouble with asserting the *grey* disabled colour directly is that the grey is itself a design choice that changes; what you actually care about is that the disabled button does **not** wear the active colour.

Solution

```
  Scenario: Disabled button has different background
    Given I am on "/profile"
    Then "#save-disabled" should not contain cssproperty "background-
color" with value "rgb(53, 143, 200)"
```

Useful for testing CSS state machines (`:hover`, `:focus`, `:disabled`) where the assertion is "this is not the default appearance".

This is the negation of Recipe 584: same `getComputedStyle()` read, but it passes when the computed value is *anything except* the one you named. That framing is the whole value. You are not pinning the disabled button to a specific grey, which would couple your test to a colour the designers will tweak, you are asserting it is visibly different from the active state, which is the user-facing contract. The same computed-style resolution applies, so the comparison is against the painted `rgb()` after the disabled styles have cascaded in. Negated property checks shine on CSS state machines generally: a focused input should *not* have the resting border, a `:hover` row should *not* match the default background.

Pro Tip: a negated check is looser by nature, `should not contain cssproperty "background-color" with value "rgb(53, 143, 200)"` would pass even if the disabled button rendered hot pink. That is the right trade for asserting *difference* between states, but it means you

should reach for the positive assertion of Recipe 584 for the one or two colours that are genuine brand contracts, and lean on negation everywhere the claim is really “not the default appearance”.

Recipe 586 Spatial Relations Between Elements

Problem

A CSS change to the notification container moved success toasts from the top of the screen to the bottom, behind the sticky footer, where nobody saw them, so users kept re-submitting forms that had already succeeded. Every text assertion still passed; the toast existed, it said the right thing, it was simply in the wrong place. You want a test that asserts *where* an element renders, not just that it exists.

Solution

```
  Scenario: Toast position
    Given I am on "/"
    When I click "Save"
    Then the element ".toast" should be at the top of the viewport within
2 seconds
```

Other spatial steps include `centered in the viewport`, `displayed within a viewport`, `should appear after the text "X"`. Useful for layout tests where the assertion is “this is in the expected position” rather than “this text exists”.

Position steps work off the element’s bounding box, Playwright’s `boundingBox()` returns the element’s `x`, `y`, `width`, and `height` relative to the viewport, and the step compares those numbers against the region you named. “At the top of the viewport” means the box sits within a band of the viewport’s top edge; the registry’s `offset` (default 60px, `WEBSHIP_SELECTORS_OFFSET`) is the tolerance that keeps a few pixels of slack from failing the check. The `within 2 seconds` makes it a web-first assertion: it re-reads the bounding box on a

poll until the toast has actually animated into place, so a slide-in transition doesn't race the check. Because it reads geometry, it catches exactly the class of bug text assertions miss, the element is present and correct but laid out wrong.

At scale: spatial assertions are sharpest when paired with the viewport sizing from Recipe 590, because "where does this render" has a different answer at every breakpoint. The pattern that pays off across many projects: a small set of layout smoke checks, toast at top, primary CTA above the fold, sticky header pinned, run as a `Scenario Outline` over your breakpoints, so a CSS change that only breaks the layout at `md` gets caught by the same handful of steps everywhere. Tune the `WEBSHIP_SELECTORS_OFFSET` tolerance once at the suite level rather than sprinkling pixel fudge factors through features; a single env-var value keeps every project's idea of "at the top" consistent and tweakable from CI without a code change.

Recipe 587 Element Ordering

Problem

A template change reversed the breadcrumb trail, the current page rendered first and "Home" last, and every link still worked, so it shipped. Users navigating by breadcrumb ended up one level too deep and complained the site was "confusing". Order is content too, and you want a test that asserts the trail reads Home, then Articles, then the title, in that sequence.

Solution

```
Scenario: Breadcrumb order
  Given I am on "/articles/42"
  # NOTE: the line below is intentionally wrong -- it claims Home comes
  # after Articles. Read the direction of "appear after" carefully:
  #   Then the text "Home" should appear after the text "Articles"
  # The correct assertions, top of trail first, are:
  Then the text "Articles" should appear after the text "Home"
  And the text "My Article" should appear after the text "Articles"
```

Ordering steps work on text node order in the DOM. They catch breadcrumb regressions, list-sort bugs, and table-row ordering issues.

The direction trips people up, so read it literally: `"Articles" should appear after the text "Home"` passes when “Articles” comes *later* in document order than “Home”, so the top of the breadcrumb is the text you name *second*. Under the hood the step finds both text nodes and compares their position in the DOM using the document’s natural traversal order (the same order `compareDocumentPosition` reports), so it reflects source order, not visual order. That distinction matters: CSS flexbox `order` or `row-reverse` can paint elements in a different sequence than they sit in the DOM, and this assertion follows the DOM. For a breadcrumb that is exactly right, you want to know the markup is in the correct order so it reads correctly to a screen reader, regardless of styling.

At scale: ordering assertions are the cheapest guard against a whole family of regressions that text checks miss, breadcrumbs reversed, a sorted list that quietly stopped sorting, table rows in the wrong sequence after a query change. Across many projects the high-value targets are the same: breadcrumbs, sorted result lists, step indicators, navigation menus. The catch to flag in review is the DOM-versus-visual gap, if a project uses flexbox reordering for responsive layouts, an ordering assertion can pass while the rendered order is wrong, so reserve these for content where DOM order *is* the contract (accessibility, semantic sequence) and lean on spatial checks (Recipe 586) when the visual position is what matters.

Recipe 588 Top-Offset Asserts

Problem

The sticky header that holds the search box and cart icon stopped sticking after a `position` change in a shared layout component, it scrolled away with the rest of the page, so on long articles the cart was unreachable without scrolling all the way back up. It looked perfect on a short page, which is how it passed review. You want a test that scrolls down and then checks the header is still pinned to the top.

Solution

```
  Scenario: Header stays pinned
    Given I am on "/long-article"
    When I scroll to bottom
    Then the element ".sticky-header" should be displayed within a
viewport with a top offset of 0 pixels
```

The top-offset variant asserts the element's vertical position relative to the viewport top. Use 0 for sticky headers, negative values for off-screen modals.

This builds on the bounding-box read from Recipe 586 but pins one number: the box's `y` relative to the viewport top, after the scroll. `I scroll to bottom` moves the page as far down as it goes, then the assertion re-reads the header's position, and a genuinely sticky header reports `y` near 0 because it stays glued to the top edge while the document slides under it. A header that lost its stickiness scrolls off and reports a large negative `y` (or vanishes from the box entirely), so the check fails. The registry's `offset` (default 60px, `WEBSHIP_SELECTORS_OFFSET`) is the tolerance band, so "a top offset of 0 pixels" forgives the handful of pixels a real browser rounds to. Negative offsets let you assert the opposite, an off-screen drawer parked just above the fold. Sticky regressions are a classic shared-component failure that a short-page smoke test never catches, so this one short scenario, run over your breakpoints (Recipe 590), pays for itself many times over, it catches a bug that otherwise surfaces as scattered "I can't find the cart" tickets.

Pro Tip: keep the tolerance in the shared `WEBSHIP_SELECTORS_OFFSET` env var rather than fudging pixel values per feature. A single value means every project agrees on what "pinned to the top" means, and you can widen it once from CI if a particular browser image rounds positions differently, without editing a single feature file.

Recipe 589 Setting Custom Breakpoints

Problem

Your design system breaks at `md: 768` and `lg: 1024`, but webship-js defaults to `md 768 / lg 992`, so a responsive test that "passed at lg" was actually testing 992px, a width your layout never targets, and missed a bug that only shows between 992 and 1024. You want your suite's breakpoint names to mean the same pixel widths your CSS does, defined once for the whole project.

Solution

Configure breakpoints in `worldParameters`:

```
// cucumber.js
module.exports = {
  default: {
    worldParameters: {
      breakpoints: {
        xs: 480,
        sm: 640,
        md: 768,
        lg: 1024,
        xl: 1280,
        xxl: 1600,
      },
    },
  },
};
```

Then use them in features:

```
Scenario: Mobile layout
  Given I am viewing the site on a "mobile" device
  And I am on "/"
  Then ".desktop-nav" should be hidden
```

Or directly via pixel size:

```
When I set the viewport to the "md" breakpoint
```

The named breakpoints make features readable; the pixel sizes make them precise.

`worldParameters.breakpoints` overrides the registry's built-in map, the shipped defaults are `xs 375x667 / sm 576x800 / md 768x1024 / lg 992x768 / xl 1200x900 (default) / xxl 1400x900 / xxxl 1920x1080`, and your config replaces the widths so a name resolves to *your* design system's pixels. When a step says `set the viewport to the "md" breakpoint`, the resolver looks up `md` in this map and calls Playwright's `page.setViewportSize()` with the matching dimensions; from then on every layout, visibility, and spatial assertion is measured at that width. That is the whole point of naming them: `md` in a feature file reads as intent, and the one number behind it lives in config where a designer can confirm it matches the CSS. The same map can be supplied at runtime via the `WEBSHIP_SELECTORS_BREAKPOINTS` env var as JSON, which matters for the next paragraph.

At scale: breakpoints are configuration, not test logic, so they belong in `worldParameters` (or the `WEBSHIP_SELECTORS_BREAKPOINTS` env var), never hard-coded as raw pixel sizes inside features. Set it via the env var and CI can run the *same* feature files against a tablet-first profile one night and a desktop-first profile the next, no feature edits, exactly what you want when a fleet of products share scenario templates but not a design system. Pin the breakpoint map per project in version control so "lg" means one thing in that suite forever; the alternative, some features at named breakpoints, some at raw pixels, is how you end up testing a width your CSS never uses and trusting a green run that proved nothing.

Recipe 590 Multi-Viewport Smoke Sweep

Problem

The hero image looked great on every laptop the team owns and vanished entirely at one specific tablet width, because a media query had an off-by-one boundary. Nobody tested that exact width by hand, who would?, and it shipped, and the first you heard was a customer screenshot of an empty hero. You want one test that checks the page holds together at every breakpoint, not just the one on your monitor.

Solution

```
Scenario Outline: Landing renders at <breakpoint>
  Given I am on "/"
  When I set the viewport to the "<breakpoint>" breakpoint
  Then ".hero" should be visible within 1 second
  And there should be no JavaScript errors

  Examples:
    | breakpoint |
    | xs         |
    | md         |
    | lg         |
    | xl         |
```

One outline, four breakpoints, four smoke checks. Catches the bug where the hero disappears at one specific viewport size.

A `Scenario Outline` expands into one real scenario per `Examples` row, so this is four independent runs, fresh context each, fresh breakpoint each. `set the viewport to the "<breakpoint>"` resolves the name through the breakpoint map from Recipe 589 and resizes the page; `".hero" should be visible within 1 second` is a web-first check that polls for up to a second, so a hero that fades in at one width still passes. The `there should be no JavaScript errors` step leans on webship-js's error capture (`WEBSHIP_JS_ERROR_MODE`, default `warn`), it

asserts no console errors fired while the page rendered at that size, which catches responsive code that throws only at certain widths. Together they are a real smoke test: the page renders *and* doesn't error, at every breakpoint, from one short outline.

At scale: the multi-viewport outline is one of the highest-value patterns you can standardize across a fleet. Template it once, hero visible, no JS errors, primary nav reachable, and drop it into every product suite with only the selectors swapped, and you have a comparable responsive-health signal across 20 sites that a PMO can actually read row by row. Mind the CI cost, though: every row is a full scenario, so a five-breakpoint sweep over ten pages is fifty scenarios, and nightly that adds up. The usual split is the full breakpoint sweep on a nightly tag and a two-breakpoint (mobile + desktop) subset on every pull request, so PR feedback stays fast while the deep sweep still runs once a day.

Recipe 591 Cookie + Storage Hybrid Hook

Problem

The plain clear-everything hook from Recipe 577 works until you hit the one flow that genuinely needs state to survive between scenarios, a multi-step onboarding split across several scenarios for readability, where each one picks up where the last left off. The blanket hook wipes the state the next scenario needs, so you either delete the hook (and lose isolation everywhere) or fight it. You want a clean slate by default with a documented way to opt out.

Solution

```
Before({ tags: 'not @keep-state' }, async function() {
  await this.page.context().clearCookies();
  await this.page.evaluate(() => {
    try { localStorage.clear(); } catch (_) {}
    try { sessionStorage.clear(); } catch (_) {}
  });
});
```

Tag scenarios with `@keep-state` to opt out of the clean slate (rare, but useful for multi-scenario flows).

Two details make this the production-grade version of Recipe 577. First, the `{ tags: 'not @keep-state' }` filter, Cucumber only runs this `Before` for scenarios that lack the tag, so a `@keep-state` scenario skips the wipe entirely. That is the opt-out: explicit, visible in the feature file, and greppable. Second, the `try/catch` around each `clear()`, if the page is still on `about:blank` (no origin storage, throws on access, the same trap Recipe 574 and 575 describe), the catch swallows it and the hook moves on instead of failing the run. The cookie clear needs no guard because `context.clearCookies()` is context-level and safe on any page. Remember webship-js already gives every scenario a fresh context, so this is reinforcement plus the opt-out switch, not the only thing standing between you and leaked state.

At scale: the tag-gated hook is the shape that survives contact with a real multi-project estate, because it admits the truth that a few flows legitimately need shared state while keeping isolation the default everywhere else. Standardize the tag name, `@keep-state`, across all suites so it means the same thing in every repo, and make it a review trigger: a scenario carrying `@keep-state` should have a one-line comment justifying why, because nine times out of ten the real fix is to merge those scenarios into one with a clear sequence of steps rather than relying on cross-scenario order. The `try/catch` matters more here than in a single project: across a fleet, some suites start scenarios with navigation and some don't, and the guard means the *same* hook drops into all of them without one repo's `about:blank` timing breaking the build.

Recipe 592 Selector Inheritance via worldParameters

Problem

You are onboarding the twentieth project, and once again you are copying the same 20 named selectors, `main-nav`, `search-box`, `cookie-accept`, out of the last project's `tests/selectors/`. They have already drifted: three projects spell the cookie button differently, and a platform-wide markup change last month meant 20 separate edits. You want the selectors every site shares to live in one package the whole estate depends on.

Solution

Publish your selector library as an npm package:

```
{
  "name": "@yourorg/webship-selectors",
  "version": "1.0.0",
  "main": "selectors.json"
}
```

In each project's `cucumber.js`:

```
const sharedSelectors = require('@yourorg/webship-selectors');

module.exports = {
  default: {
    worldParameters: {
      selectors: {
        css: sharedSelectors.css,
        xpath: sharedSelectors.xpath,
      }
    }
  }
};
```

Now every project inherits the same selector vocabulary. Updates to the package propagate via `npm update`.

This works because `worldParameters.selectors` accepts inline `css` and `xpath` maps directly, not only `files`. The registry seeds itself from whatever you put here at world setup, so requiring the package and spreading its `css` / `xpath` objects into the config means every scenario in the project starts with the shared vocabulary already registered, identical to loading a local JSON file (Recipe 580) except the file is now a versioned dependency. You can still layer a project's own `files:` on top for that product's quirks, and inline `add` steps still override per-scenario; the precedence is the same as everywhere else in the registry. The package's `main` points at the JSON, so `require()` hands you a plain object with `css` and `xpath` keys, the same shape the registry expects.

At scale: this is the recipe the whole chapter has been building toward for a multi-project team. A versioned selector package turns "the login button" from a string copied 20 times into a single source of truth the whole estate consumes, and `npm`'s semver gives you the release control that copy-paste never could. A platform-wide markup change becomes one PR to the package and a `^1.x` bump that flows out as suites update, instead of 20 hand-edits that miss two repos. Two practices keep it healthy: version it like real code (a selector rename is a minor bump, a removal is a breaking major), and keep the package to genuinely *shared* selectors, the nav, search, consent, footer that every site has, while each product's idiosyncratic selectors stay in its own local file. That split is what lets DevOps pin a known-good selector version per environment and lets a PO trust that "search-box" means the same element in every project's report.

Recipe 593 Cookie-Driven Feature-Flag Test

Problem

The team is rolling out a new checkout behind a `flags.new-checkout` cookie, on for 10% of traffic, off for the rest. QA needs both branches tested every build, but the old approach was to ask DevOps to flip the flag in a config file, run the suite, flip it back. That is a ticket, a wait, and a flag that occasionally got left in the wrong state. You want one suite that exercises both branches without anyone touching server config.

Solution

```
Scenario: Old checkout
  Given the cookie "flags.new-checkout" is set to "off"
  When I am on "/checkout"
  Then ".checkout-v1" should be visible within 2 seconds

Scenario: New checkout
  Given the cookie "flags.new-checkout" is set to "on"
  When I am on "/checkout"
  Then ".checkout-v2" should be visible within 2 seconds
```

Two scenarios cover both branches without modifying server-side configuration. Useful during a gradual rollout.

Each scenario plants the flag cookie before navigating, the Recipe 566 trick, so the server sees it in the `Cookie` header of the first request and renders the right branch on first paint. This only works because the flag is read from a *cookie*; the value is in the jar before the page loads, no clicking through a settings UI to toggle it. Two separate scenarios, two fresh contexts, so the `off` run can't bleed into the `on` run, the framework's per-scenario context guarantees each branch starts clean. The web-first `within 2 seconds` on each side absorbs whatever async work the chosen checkout does before its root element appears.

At scale: cookie-driven flag testing is how you keep both sides of a rollout green across a fleet without coupling your suite to server state. The pattern that holds: one cookie per flag, both branches tested, and the flag cookie set in the feature rather than in environment config, so the same suite proves the new and old paths on every environment regardless of what percentage the real flag is dialed to. Tag the new-branch scenarios (`@flag-new-checkout`) so DevOps can include or exclude in-flight features per environment, and so the day the flag is removed you can grep the tag to find every scenario that needs cleaning up. The alternative, testing only whichever branch the environment's flag happens to expose, means your green run silently stops covering the other half the moment someone changes a percentage.

Recipe 594 Storage-Driven Multi-Step Wizard Test

Problem

A five-step signup wizard saves progress to localStorage so users can leave and come back. The "returns mid-wizard" path is exactly where bugs hide, a restored field that doesn't repopulate, a step that re-asks a question, but the only way the team tested it was to fill steps 1 through 3 every run, three minutes of setup to reach the ten seconds that mattered, and the setup itself broke whenever step 2's form changed. You want to land on step 3 with real data already in place.

Solution

```
  Scenario: Wizard restores from saved state
    Given the local storage "wizard.step" is set to "3"
    And the local storage "wizard.data" is set to "
{\"name\":\"Sara\",\"email\":\"sara@example.com\"}"
    When I am on "/wizard"
    Then ".wizard-step-3" should be visible within 2 seconds
    And "input[name=name]" should have value "Sara"
    And "input[name=email]" should have value "sara@example.com"
```

The seeded storage state makes the wizard think the user is mid-flow. Saves you from running steps 1-3 to test step 3.

This is Recipe 574's storage seeding doing real work, and the JSON line is the detail to study. localStorage only stores strings, the step does `localStorage.setItem('wizard.data', value)` and the value must already be a string, so the object is hand-serialized in the Gherkin, with `\"` escaping the inner quotes (Cucumber's `{string}` honours backslash-escaped quotes, so the app reads back valid JSON it can `JSON.parse`). Note the step order looks like seed-then-navigate, but the working sequence is the one Recipe 574 spells out: navigate so the origin exists, the keys land, the wizard reads them on load. The three assertions then confirm the restore actually worked end to end, the right step is showing *and* the fields repopulated, which is the whole point, because a wizard that jumps to step 3 with empty fields is its own bug.

At scale: seeding storage to jump to a deep state is the single biggest runtime saving you can template across a fleet of wizard-heavy products. Every multi-step flow, checkout, onboarding, application forms, has the same shape, and across 20 suites the difference between "fill three steps to test the fourth" and "seed and land" is minutes per scenario multiplied by every run, every night. The discipline that keeps it honest: keep at least one *full* walk-through scenario that actually clicks through every step (so the seeding can't mask a broken step transition), and use the seeded shortcut for the many variations of "what happens when you return at step N". Store the seed payloads as named fixtures rather than inline JSON repeated across scenarios, so a schema change to `wizard.data` is one fixture edit, not a sweep through every feature that hard-codes the blob.

Recipe 595 Asserting Cookie Path/Domain

Problem

Consent has to be remembered across `www.example.com`, `shop.example.com`, and `help.example.com`, which means the consent cookie must be set on the apex `.example.com` domain. A refactor set it on `www.` instead, so users re-saw the banner on every subdomain and the analytics team flagged a sudden spike in consent prompts. The twelve built-in matchers check name and value, not domain, so this is the case where you write a small custom step against the cookie object itself.

Solution

Use the world helper:

```
Then(/^the cookie "([^"]*)" should be on domain "([^"]*)"$/, async
function(name, domain) {
  const cookies = await this.page.context().cookies();
  const found = cookies.find(c => c.name === name);
  if (!found) throw friendly(`No cookie named "${name}".`);
  if (found.domain !== domain) {
    throw friendly(`Cookie "${name}" is on domain "${found.domain}",
expected "${domain}".`);
  }
});
```

Then in features:

```
  Scenario: Consent cookie is domain-wide
    Given I am on "https://www.example.com/"
    When I click "Accept all"
    Then the cookie "consent" should be on domain ".example.com"
```

This reaches for the same `context.cookies()` jar the built-in matchers read, but inspects a field they don't expose, the `domain`. Every object that call returns carries the full cookie: `name`, `value`, `domain`, `path`, `expires`, `httpOnly`, `secure`, `sameSite`. The built-in twelve matchers only branch on `name` and `value`; when you need to assert *scope*, domain or path, a six-line custom step against the cookie object is the clean way to do it, and it sits right alongside the standard matchers since they share the data source. The `friendly()` helper is webship-js's error wrapper; it produces the readable `expected vs. actual` failure the built-ins give you, so a domain mismatch reports `is on domain "www.example.com", expected ".example.com"` rather than an opaque assertion error. The same shape extends to `path`, swap `domain` for `path` and you have a path-scope assertion.

At scale: apex-versus-subdomain cookie scope is a bug that only surfaces in production-shaped, multi-subdomain environments, it passes every single-host test and breaks the moment real traffic crosses subdomains, which is precisely the kind of regression a fleet needs to catch *before* release, not after. If more than one project asserts cookie scope, promote this custom step into your shared step-definitions package so every suite phrases it identically and review can read it without re-deriving the regex. And echo the built-in matchers' lesson here: assert the *least specific* domain that still encodes the contract, pin `.example.com` when subdomain-sharing is the requirement, but don't over-assert a `path` the app is free to change.

Recipe 596 IndexedDB Cleanup (Custom)

Problem

Your PWA caches the user's data in IndexedDB so it works offline, and the offline tests passed individually but flaked when run together, an earlier scenario's IndexedDB records bled into a later one that expected an empty store, so a "first sync" test found data already there. Webship-js clears cookies and localStorage between scenarios but doesn't touch IndexedDB, and it ships no step for it. You want offline scenarios to start with an empty database.

Solution

Custom step:

```
Before({ tags: '@offline' }, async function() {
  await this.page.evaluate(async () => {
    const dbs = await indexedDB.databases();
    await Promise.all(dbs.map(db => new Promise((resolve) => {
      const req = indexedDB.deleteDatabase(db.name);
      req.onsuccess = req.onerror = resolve;
    })));
  });
});
```

Now scenarios tagged `@offline` start with empty IndexedDB.

Here is the gap this fills. The framework's fresh-context guarantee covers cookies and storage, but IndexedDB is a third storage layer and its lifetime is its own, so it is the one piece of browser state that can survive between scenarios if your app writes to it. The hook runs `indexedDB.databases()` inside the page to enumerate every database for the current origin, then deletes each one, wrapping the delete in a promise because IndexedDB's API is event-based (`onsuccess` / `onerror`) rather than promise-based, the `resolve` on both events means a delete that errors doesn't hang the hook. Note the `{ tags: '@offline' }` scope: this only runs for offline scenarios, because tearing down IndexedDB has a cost and most scenarios never touch it. Same per-origin rule as localStorage, it clears the current origin's databases, nothing on another subdomain.

At scale: IndexedDB, service workers (next recipe), and the HTTP cache are the three state layers that the standard isolation doesn't reach, and across a fleet of PWAs they are the source of the flakiness that "passes alone, fails in the suite". Bundle the cleanup behind a single capability tag, `@offline` or `@pwa`, and put the hook in your shared hooks package so every PWA project gets identical teardown without re-implementing the event-to-promise dance. The tag scoping is the cost control: PWA teardown is heavier than a cookie clear, so a marketing-site suite with no offline behaviour never pays for a layer it doesn't use, while the PWA suites get bulletproof isolation on the scenarios that need it.

Recipe 597 Service Worker Reset

Problem

A service worker caches the app shell, which is great for returning users and a nightmare for testing the *first* visit, a registered service worker can serve a stale cached shell, so your "first visit shows the install prompt" test sometimes saw the cached version with the prompt already dismissed. It passed or failed depending on what the previous scenario left registered. You want first-visit scenarios to start with no service worker at all.

Solution

```
Before({ tags: '@first-visit' }, async function() {
  await this.page.evaluate(async () => {
    const regs = await navigator.serviceWorker.getRegistrations();
    await Promise.all(regs.map(r => r.unregister()));
  });
});
```

Pair with `Before { @first-visit }` to ensure both the SW and the cookie jar are clean for those scenarios.

Service workers are the second state layer the fresh context doesn't fully reset, a worker registered in one scenario can still be controlling the page in the next, and it intercepts requests invisibly, which is what makes it the sneakiest source of "works locally, flakes in CI". The hook runs inside the page, asks the browser for every registration on the origin via `navigator.serviceWorker.getRegistrations()`, and calls `unregister()` on each, awaiting all of them so the page is genuinely worker-free before the scenario's first navigation. That ordering is the point: unregister *before* you navigate, so the first request actually hits the network instead of a cached shell the old worker would have served. The `{ tags: '@first-visit' }` scope keeps this off the scenarios that don't care, and the per-origin rule applies as always, you are clearing workers for the current origin only. The tags compose: put `@first-visit @offline` on one scenario and Cucumber runs both this hook and Recipe 596's IndexedDB reset, giving you a

genuinely cold browser, the only honest way to test install prompts, onboarding, and offline-fallback behaviour. Ship both hooks from the shared package and “truly first visit” means the same thing in every product without a bespoke reset per repo.

Recipe 598 Network Cache Bypass

Problem

A performance regression test for cold start kept reporting suspiciously fast load times, because the browser was serving assets from cache that a real first-time visitor would download from scratch, so the test “passed” while a genuine cold start was twice as slow. You want every asset fetched fresh, the way a brand-new visitor’s browser would, so the cold-start measurement means something.

Solution

Configure the context:

```
this.context = await browser.newContext({
  // No setting needed for cache -- Playwright contexts start with empty
cache by default.
  // But you can force bypass per-request:
  extraHTTPHeaders: { 'Cache-Control': 'no-cache' },
});
```

Each scenario gets a fresh context, so the cache is per-scenario by default. The header forces cache bypass on the network layer.

The HTTP cache is the third state layer beyond cookies and storage, and here the framework already helps more than you’d expect: a fresh Playwright browser context starts with an empty cache, and webship-js gives every scenario its own context, so scenario-to-scenario, the cache is clean without you doing anything. What this recipe adds is bypass *within* a scenario that loads the same assets more than once. Setting `extraHTTPHeaders: { 'Cache-Control': 'no-cache'`

`}` on the context attaches that header to every request the context makes, telling the network layer to revalidate rather than serve from its own cache. That is what keeps a multi-navigation cold-start measurement honest, the second visit in the same scenario still fetches fresh instead of pulling the first visit's assets from memory.

At scale: the cache is the quietest of the three extra state layers, which is exactly why it skews performance numbers without anyone noticing. Across a fleet, performance and cold-start tests are only comparable between projects if they all define "cold" the same way, empty cache, no service worker (Recipe 597), no IndexedDB (Recipe 596). Treat the trio as one capability: a `@cold-start` tag that wires up all three, shipped from the shared hooks package, so a load-time number from project A means the same thing as one from project B and a PMO can actually compare them. The fresh-context default does the heavy lifting; the explicit header is for the in-scenario re-fetch case, and worth a comment in the feature so the next reader knows why it's there.

Recipe 599 Hybrid Cookie + Selectors Pattern

Problem

Premium subscribers see a different dashboard, and testing it used to mean a real premium login, a fixture account, credentials in the feature file, a slow trip through the login form, just to verify one banner renders. Free and premium had separate feature files that drifted apart, and the premium one broke every time the login flow changed. You want one readable scenario that puts the browser in the premium tier and checks the premium UI, no login, no duplicated file.

Solution

```
@premium
Scenario: Premium dashboard
  Given the cookie "tier" is set to "premium"
  And I add "premium-banner" selector for "section[data-tier='premium']"
css selector
  When I am on "/dashboard"
  Then "premium-banner" should be visible within 2 seconds
```

Cookie sets the state, named selector keeps the assertion readable. The combination scales to dozens of tier-aware scenarios.

This is the chapter's two halves working together. The cookie step (Recipe 566) plants `tier=premium` in the context before navigation, so the server renders the premium dashboard on first paint, no login, because the tier is carried by a cookie the test controls. The named-selector step (Recipe 578) registers `premium-banner` in the scenario's registry, so the assertion reads as the intent (`"premium-banner" should be visible`) rather than the raw `section[data-tier='premium']`. The web-first `within 2 seconds` polls until the banner appears. State and vocabulary are decoupled: the cookie decides *which* UI renders, the selector names *what* you check, and because both are per-scenario, the free-tier scenario right next to it sets `tier=free` and registers nothing premium, with zero risk of bleed.

At scale: tier-, role-, and flag-aware UIs multiply fast across a fleet, free/premium/enterprise times a dozen products is a lot of variants, and the cookie-plus-selector pattern is what keeps that combinatorial space testable. Set the tier with a cookie instead of a login, and a tier change is one line, not a fixture account per tier per environment. Pull the tier selectors into a shared JSON file (Recipe 580) so `premium-banner` means the same element everywhere, and tag by tier (`@premium`, `@enterprise`) so CI can run a single tier's scenarios on demand and reports can break quality down by tier. The anti-pattern this replaces, a logged-in fixture account per tier, duplicated feature files that drift, is exactly the maintenance load that sinks a multi-project suite over time.

Recipe 600 The State-Setup Hook Pattern

Pattern

By the time you have onboarded a few projects, you notice the same three lines of setup at the top of nearly every scenario, clear state, ensure fixtures, register selectors, copied with small variations, drifting apart, each one a place a future flake can hide. The fix is to stop repeating it in features and put it in one hook. Most tests need three things at the start:

1. Clear cookies, localStorage, sessionStorage
2. Set up any test-data fixtures
3. Register any feature-specific named selectors

A global Before hook handles all three:

```
const fs = require('fs');

Before(async function() {
  // 1. Clean state
  await this.page.context().clearCookies();
  await this.page.evaluate(() => { localStorage.clear();
sessionStorage.clear(); });

  // 2. Fixtures (idempotent -- only inserts missing rows)
  await this.installFixtures();

  // 3. Selectors
  const selectorFiles = ['tests/selectors/global.json',
'tests/selectors/forms.json'];
  for (const file of selectorFiles) {
    if (fs.existsSync(file)) {
      const data = JSON.parse(fs.readFileSync(file, 'utf-8'));
      Object.entries(data.css || {}).forEach(([n, s]) =>
this.addCssSelector(n, s));
      Object.entries(data.xpath || {}).forEach(([n, s]) =>
this.addXpathSelector(n, s));
    }
  }
});
```

Every scenario starts identically. No surprises. Test logs are clean. The hook is the foundation, once it's right, everything else gets easier.

The hook ties the whole chapter together, one numbered concern at a time. Step 1 is Recipe 577's clean-state pair, layered on top of the framework's own fresh-context guarantee for belt-and-braces isolation. Step 2 calls a fixture installer that is *idempotent* on purpose, it only inserts rows that are missing, so it is safe to run before every scenario without piling up duplicate data or making one scenario's run depend on another's. Step 3 is Recipe 580's file loading done in code: each file is the `{"css": {}, "xpath": {}}` shape the registry expects, read fresh per scenario so nothing leaks across the boundary, with `fs.existsSync` guarding a missing file so an absent `forms.json` degrades quietly instead of crashing the hook. The result is the property every reliable suite has, every scenario starts from the exact same known state, which is what lets you write each one as if it runs first.

At scale: this hook is the contract that makes a multi-project test estate trustworthy. When the same three-part setup, clean state, idempotent fixtures, file-loaded selectors, runs identically in all 20 suites, a red scenario means a real regression, full stop, not a leaked cookie or a fixture that didn't load. Ship the hook itself from a shared package so it can't drift between repos, and let each project vary only the data inside it: its own `selectorFiles` list, its own fixture set. That separation, shared mechanism, per-project data, is the through-line of this entire chapter, and it is what turns "we test 20 sites" from a maintenance burden into a single pattern you maintain once and a quality signal a PMO, a PO, and a DevOps lead can all read the same way.

The cookie, storage, and selector subsystems aren't glamorous. They're the plumbing. But every flaky test eventually traces back to a leaked cookie, a stale storage entry, or a selector that drifted. Master the plumbing and your suite becomes orders of magnitude more reliable.

Chapter 49: Reference Patterns Library

The previous 48 chapters covered each subsystem in depth. This chapter is the **reference library**, a collection of short, self-contained patterns that span multiple subsystems. Each pattern is one or two recipes. They are not organised by feature but by the kind of problem they solve. Think of this chapter as a long appendix you read recipe-by-recipe rather than cover-to-cover.

Use the table of contents and the index to find what you need. Adopt the patterns that fit your stack. Adapt them when they don't.

Each pattern names the situation it solves, shows the shortest solution that works, and explains what the machinery underneath is doing. Where it changes the picture, an **At scale:** note covers what happens when you run the pattern across many projects at once, because most teams that need a reference library are not running one suite, they are running twenty.

Bootstrapping Patterns

Recipe 601 The Two-Line Smoke Boot

Problem: The deploy went out at midnight, and the first human to learn the homepage is broken is a customer, at 7:45 the next morning, on social media. Most outages do not need a 400-scenario suite to detect them. They need two lines that run in every pipeline, every time, and answer the only question that matters before any other test runs: is the site up, and is it healthy?

```
@smoke
Scenario: Site responds
  Given I am on "/"
  Then there should be no JavaScript errors
```

How It Works: `Given I am on "/"` resolves the path against `launchUrl` (the `LAUNCH_URL` env var), navigates, and then `smartSettle` holds the step until in-flight fetch/XHR requests drop to zero, pending timers drain, and the DOM has been quiet for at least 250ms, so a page that technically responded but is still thrashing does not slip through as a pass. The second line reads the JavaScript error buffer webship-js fills from page-error and console listeners for the whole scenario: one uncaught `TypeError` fails the check even when the page looks fine. Reachability and health are different failures, a 200 with a broken bundle passes the first line and fails the second, and that distinction is the whole recipe.

At scale: Make `@smoke` mean exactly this in every project you run, and wire it into every pipeline on every push. When twenty sites share the convention, “is everything up?” stops being twenty Slack questions and becomes one CI dashboard filter. The cost is seconds per project; the payoff is that nobody has to ask.

Recipe 602 The Status-Sentinel

Problem: The frontend smoke is green, but the mobile app is down, because the API died and nothing tests the API on its own. Backend health gets discovered by whoever consumes it first, and that is usually not your team. One scenario per critical endpoint closes the gap for the cost of three HTTP requests.

```
@smoke
Scenario Outline: <endpoint> responds healthy
  When I send a GET request to "<endpoint>"
  Then the API response code should be 200

  Examples:
    | endpoint        |
    | /api/health     |
    | /api/version    |
    | /api/ping       |
```

How It Works: The API steps run on axios, not on the browser, no page, no rendering, no settle logic, so each probe costs milliseconds. The Scenario Outline expands into one independent, named scenario per row: a dead endpoint fails as `/api/version responds healthy` in the report instead of hiding inside a loop. When the API lives on a different host than the site, set it once with `Given the API base URL is "..."` and keep the table paths relative.

At scale: Standardize the endpoint names before you standardize the test, `/api/health` on every service, not `/healthz` here and `/status` there. Once the convention holds, the cucumber JSON from twenty projects answers “which services are degraded right now?” with one query, which is the first portfolio-level quality signal most delivery managers ever get.

Recipe 603 The Pre-Test Database Seed

Problem: Half your scenarios need articles to exist; the other half could not care less. Seed everything for everyone and the suite crawls. Seed nothing and the listing scenarios pass vacuously on a fresh database, right up until the new hire clones the project, runs the suite against a clean environment, and spends an afternoon discovering that "no results found" was never supposed to be green.

```
Before({ tags: '@with-articles' }, async function() {
  await this.api.post('/api/test/seed/articles', { count: 5 });
});
```

How It Works: Cucumber's `Before({ tags })` option takes a tag expression and runs the hook only for matching scenarios, everything else skips the cost entirely. The hook calls a test-only seeding endpoint through the world's API client before the browser does anything, so the data exists before the first navigation. Because the seed is per-scenario rather than per-suite, ordering does not matter and parallel shards never fight over shared fixtures. Scenarios declare their data needs in their tags, which means the feature file tells the whole truth.

At scale: Standardize the tag vocabulary and the endpoint shape across projects, `@with-articles`, `@with-users`, `/api/test/seed/*`, and guard the seed routes so they only exist outside production. A QA lead who owns twenty suites should be able to read any scenario's tags and know its data requirements without opening a single hook file. That convention is worth more than any individual seed.

Authentication Patterns

Recipe 604 The Auth Storage State

Problem: Two hundred scenarios need an admin, and each one starts with the same four steps: open the login page, type, type, click. That is five seconds of pure overhead per scenario, fifteen minutes of login theatre per run, and every one of those logins is a fresh chance for a slow auth service to fail a test that was never about login. Log in once, save the proof, and let every scenario start already inside.

```
// global-setup.js
await page.goto('/user/login');
await page.fill('#username', 'admin');
await page.fill('#password', 'admin-pass');
await page.click('#submit');
await page.context().storageState({ path: 'tests/auth/admin.json' });
```

Then in a Before hook:

```
Before({ tags: '@as-admin' }, async function() {
  await
this.context.addCookies(require('./tests/auth/admin.json').cookies);
});
```

How It Works: `storageState()` serialises everything the browser holds that makes you "logged in", cookies plus localStorage, into one JSON file. The Before hook injects those cookies into the scenario's fresh context through `this.context.addCookies()` before the first navigation, so the very first page load arrives already authenticated. The tag filter means only `@as-admin` scenarios pay even that tiny cost; anonymous scenarios stay anonymous. webship-js 2.0 also ships this as built-in steps, `auth.steps.js` has save/restore/clear storage-state steps, so you can run the whole dance from Gherkin without writing the hook yourself. One thing to watch: saved sessions expire. Regenerate the JSON at suite start, not once a sprint, or you will chase phantom 403s.

At scale: Never commit those JSON files, they are credentials. Generate them in CI from secrets (admin user and password as env vars per environment) in a setup job, and give every project the same `tests/auth/` layout so the hook code is copy-paste identical across all twenty repos. When a login flow changes in one product, you fix one global-setup file, not two hundred scenarios.

Recipe 605 The Role-Based Storage State

Problem: The bug that reaches production is rarely "admin can't do X", admins can do everything, so admin-only suites stay green while editors, reviewers, and plain readers hit walls nobody tested. The day you find out is the day a real reviewer cannot approve anything. Testing permission boundaries means running the same flows as four different people, and nobody wants four copies of the login dance.

One file per role:

```
tests/auth/
  admin.json
  editor.json
  reviewer.json
  reader.json
```

Tag-driven:

```
['admin', 'editor', 'reviewer', 'reader'].forEach(role => {
  Before({ tags: `@as-${role}` }, async function() {
    await
this.context.addCookies(require(`./tests/auth/${role}.json`).cookies);
  });
});
```

How It Works: The `forEach` registers four separate `Before` hooks, one per tag expression, at load time, Cucumber treats them as four independent hooks, and only the one matching the scenario's tags fires. A scenario tagged `@as-editor` gets exactly the editor's cookies injected

into its fresh context; an untagged scenario gets nobody's. The role list lives in one line, so adding a fifth role is a one-word change plus one more JSON file. The real value shows in review: `@as-reviewer` at the top of a scenario documents whose eyes the test looks through, without a single login step in the feature file.

At scale: Agree on the role vocabulary once, `@as-admin`, `@as-editor`, `@as-reviewer`, and use it in every project, even when the underlying products call the roles different things. A QA lead skimming an unfamiliar suite should never need to ask what `@as-editor` means. It also makes portfolio questions answerable: "show me every scenario we run as an anonymous user" becomes a tag grep across repos instead of an archaeology project.

Recipe 606 The Multi-Factor Mock

Problem: Security turned on MFA for the staging environment, good for security, fatal for automation. The suite cannot read a phone, and the TOTP secret the team shared in a chat thread will rotate the week everyone forgot the tests depend on it. You still need to test everything that lives behind the login, so mock the one request that checks the second factor and walk through the rest of the flow for real.

```
Scenario: Login with mocked MFA
  Given the URL "**/api/mfa/verify" returns the JSON:
    """
    { "verified": true }
    """
  When I am on "/user/login"
  And I fill in "username" with "admin"
  And I fill in "password" with "secret"
  And I press "Log in"
  And I fill in "code" with "000000"
  And I press "Verify"
  Then "/dashboard" should be the current path
```

How It Works: The mock step registers a `page.route()` handler against the `**/api/mfa/verify` glob, the `**` matches any scheme and host, so the same pattern works on localhost and on staging. From that moment, any request the verification form sends to that endpoint never leaves the browser: Playwright fulfils it locally with your JSON, the app reads `{ "verified": true }`, and the flow proceeds. Everything else, the username/password check, the redirect logic, the session cookie, is the real application. You are not testing MFA itself here (do that in a dedicated lane with a test TOTP seed, or by hand); you are stopping MFA from blocking the two hundred scenarios that live behind it.

At scale: Decide per portfolio where MFA gets mocked and where it stays real, and write the decision down, one project mocking silently while another maintains a TOTP seed is how a real MFA breakage hides for a month. A shared `@mfa-mocked` tag on these scenarios keeps the trade-off visible in every report.

Layout and Visual Patterns

Recipe 607 The Sticky-Header Guard

Problem: The sticky header worked for a year, then a refactor wrapped it in a parent with `overflow: hidden`, and it quietly stopped sticking. No error, no broken layout, nothing a smoke test notices: the nav simply scrolls away, and users on long articles lose the menu. CSS regressions like this are invisible to functional tests unless you assert the behaviour itself.

```
Scenario: Header stays pinned on scroll
  Given I am on "/long-article"
  When I scroll to bottom of the page
  Then the element ".sticky-header" should be at the top of the viewport
```

How It Works: The scroll step drives the page to the bottom, and the viewport assertion reads the element's `boundingBox()` and requires its top edge to still sit at (or near) the top of the viewport after scrolling, that is the operational definition of "sticky". A plain visibility check would pass either way, which is why this recipe exists. The subtlety worth knowing: `position: sticky` fails silently when any ancestor has `overflow: hidden`, the CSS still parses, the computed style still says sticky, but the box scrolls away. That is exactly the class of regression this guard catches.

At scale: Sticky regressions ride in on shared layout components, a theme bump, a wrapper added in a design-system release, so they hit every project on that stack at the same time. Point `.sticky-header` at a named entry in a shared `tests/selectors/` JSON file instead of hard-coding the class in each repo, and the day the markup changes you edit one file, not twenty suites. The check costs a scroll and one box read, so run it once per layout rather than once per page.

Recipe 608 The Footer-Anchor Test

Problem: Short pages are where footers betray you. The contact page has three lines of content, and the footer floats mid-screen with a desert of white space below it, the kind of bug every visitor notices and no suite catches, because the footer is technically present, technically visible, and technically wrong. It usually ships inside a layout-wrapper refactor, and nobody scrolls a short page again until a customer does.

```
Scenario: Footer is at the bottom on short pages
  Given I am on "/contact"
  Then the element "footer" should be displayed within a viewport
```

How It Works: `should be displayed within a viewport` asserts the element actually sits inside the visible viewport rectangle, not merely attached to the DOM or un-hidden by CSS. On a short page with a correct min-height layout (a flex column or grid pinning the footer), the footer lands inside the first viewport without scrolling, so the assertion passes; a footer that collapsed upward or got pushed below a phantom spacer fails it. Pick your shortest real page for this test, the failure mode only exists where content does not fill the screen.

At scale: The footer markup is almost always shared, one template, every page, every site on the stack, so this is a once-per-project check, not a per-page sweep. Keep the shortest-page path in the same env-var or config slot you use for other layout checks (a thin 404 or contact page works on most sites) so the same scenario drops into any repo without edits. When it fails, it fails everywhere that shares the template, which is the signal you want.

Recipe 609 The Print Stylesheet Test

Problem: Somebody prints your article, a lawyer, a professor, someone filing paperwork, and gets three pages of navigation menus, cookie banners, and a hamburger icon rendered as a stray glyph. Print stylesheets break silently because nobody on the team has printed a page since the stylesheet was written, and the regression usually ships inside an unrelated CSS refactor.

```
Scenario: Print view hides nav
  Given I am on "/articles/42"
  When I set the viewport media to "print"
  Then "nav" should be hidden
```

How It Works: The media step switches the page's emulated media type to `print`, the same engine switch the browser makes in print preview, driven by Playwright's `emulateMedia({ media: 'print' })` underneath. Every `@media print` rule applies instantly; no paper involved. The assertion then runs as normal: `"nav" should be hidden` polls the element web-first until the print stylesheet's `display: none` takes effect. The media-emulation step is not a webship-js built-in, it is a thin custom step wrapping `page.emulateMedia()`, so add it once to your project's step files. In a fuller version, assert the other direction too, the article body should remain visible, because the second-most-common print bug is a stylesheet that hides everything.

At scale: Print CSS lives in the shared theme, so one custom `set the viewport media` step published in your internal step package covers every site that uses it (see Recipe 672 on sharing steps). Most teams skip print testing entirely until a customer complains, a single emulated assertion per project, run in the nightly lane, turns "nobody has printed this in two years" into a green check.

Recipe 610 The High-Contrast Test

Problem: A user turns on Windows High Contrast or the macOS "Increase contrast" setting because they cannot read low-contrast text, and your site ignores them completely. The accessibility team flags it during an audit, weeks after the theme that broke the `prefers-contrast` hook shipped, and now it is a remediation ticket instead of a one-line catch. The preference is real, the OS exposes it, and almost nobody tests that the site responds.

```
Scenario: High-contrast preference applies
  Given I am on "/"
  When I emulate "prefers-contrast: more"
  Then "body" should have class "contrast-high" within 1 second
```

How It Works: `emulate "prefers-contrast: more"` is a custom step wrapping Playwright's `page.emulateMedia({ contrast: 'more' })`, it sets the media feature the OS would set, so any `@media (prefers-contrast: more)` rule fires exactly as it would for a real user, no OS toggling required. The assertion then polls web-first (100ms, capped here at `within 1 second`) until the contrast hook adds its class. You are testing one thing: that the site notices the preference and reacts. Whether the resulting palette is *good* is a separate, human judgement, this guards the wiring.

At scale: Media-emulation steps are not built in, so write one parametrised `When I emulate "<feature>"` step that maps to `emulateMedia()` and publish it in your shared step package, the same step then powers Recipes 611 and 612 too. Across a portfolio, group the contrast, reduced-motion, and dark-mode checks under a `@preferences` tag so one filter answers "which sites respect OS accessibility settings?" without opening a single repo.

Recipe 611 The Reduced-Motion Test

Problem: A visitor with a vestibular disorder sets “reduce motion” so the web stops making them dizzy, lands on your homepage, and gets hit by the full parallax-and-carousel show anyway. For some people that is not an annoyance, it is nausea. The setting exists precisely so software can back off, and a site that ignores it is failing the users who need it most, silently, because the developers testing it never turn the preference on.

```
Scenario: Reduced motion disables animations
  Given I am on "/"
  When I emulate "prefers-reduced-motion: reduce"
  Then "html" should have attribute "data-animations" with value "off"
```

How It Works: The emulate step sets the `reduced-motion` media feature through `page.emulateMedia({ reducedMotion: 'reduce' })`, so your `@media (prefers-reduced-motion: reduce)` rules activate as if the OS asked for them. This recipe asserts the JavaScript side of the contract, a `data-animations="off"` flag the app flips when it reads the preference, which is what gates the heavy scripted animations CSS alone cannot stop. The attribute assertion polls web-first until the flag lands. Pair it with the CSS side (a transition-duration check) when motion lives in both layers.

Recipe 612 The Dark-Mode System Preference

Problem: Your site has a dark mode and a toggle for it, and everyone tested the toggle. Nobody tested the other path: a visitor whose phone is already in dark mode opening the site for the first time, expecting it to respect what the whole rest of their device is doing. When the system-preference branch breaks, usually because someone refactored the manual toggle and forgot the auto path, the first load flashes white at people who set dark mode specifically to avoid that.

```
Scenario: System dark mode triggers dark theme
  Given I am on "/"
  When I emulate "prefers-color-scheme: dark"
  Then "html" should have attribute "data-theme" with value "dark"
```

How It Works: `emulate "prefers-color-scheme: dark"` calls `page.emulateMedia({ colorScheme: 'dark' })` before navigation, so the very first paint already believes the OS is in dark mode, which is the case you are actually testing, the cold load with no manual override stored. The assertion confirms the app set `data-theme="dark"` on the root, the hook the stylesheet keys off. Test the manual toggle separately; this one guards the automatic branch that only a real OS preference triggers.

At scale: Dark mode is a shared theme feature, so the auto-detection logic is identical across every product on the stack, and so is the regression when a token refactor breaks it. One emulate step plus this three-line scenario, copied unchanged into each repo, catches it everywhere. Keep the root attribute name (`data-theme`) consistent across products so the same scenario truly is copy-paste, not a per-site rewrite.

Network and Performance Patterns

Recipe 613 The Slow-Network Smoke

Problem: The site is fast on the office fibre and unusable on a train. Your team builds on gigabit, so the page that loads instantly for you spends nine seconds blank for a customer on a congested mobile connection, and the analytics just show them bouncing. Slow-network failures are invisible to everyone who never tests on a slow network, which on most teams is everyone.

```
Scenario: Page renders on slow 3G
  Given the network is throttled to "Slow 3G"
  When I am on "/"
  Then "h1" should be visible within 10 seconds
```

How It Works: The throttle step is a custom one driving Playwright's CDP `Network.emulateNetworkConditions` (Chromium) with the standard Slow-3G profile, roughly 400kbps down and ~400ms of latency, so the request waterfall behaves like a real degraded link, not a fast one with an artificial pause. The assertion then gives the page a generous `within 10 seconds` budget and polls web-first for the `h1` to become visible. You are not measuring exact load time here; you are proving the page is still *usable*, the headline actually arrives, when the network is hostile.

At scale: Throttling needs the CDP path, which is Chromium-only, so keep these scenarios in a `@slow-network` lane that runs under `BROWSER=chromium` and skip them on firefox/webkit rather than letting them error. Across products, pick one realistic profile and one generous timeout and reuse them everywhere, a shared budget makes "is the slow-network experience getting worse?" a comparable signal instead of per-project guesswork.

Recipe 614 The Offline Smoke

Problem: You shipped a progressive web app and promised it works offline, that was the whole pitch to the client. Then a service-worker change silently breaks the offline cache, and the only way anyone finds out is a user in a tunnel staring at the browser's dinosaur instead of your app. Offline behaviour is the one feature you cannot verify by just using the site normally, because you are always online while you build it.

```
Scenario: PWA serves offline
  Given I am on "/"
  When the network is offline
  And I reload the page
  Then ".offline-indicator" should be visible
```

How It Works: Load the page online first so the service worker installs and caches its assets, that is what `Given I am on "/"` does here. Then `the network is offline` (a built-in step) calls `context.setOffline(true)`, cutting the browser off the network exactly as airplane mode would. The reload now has no network to fall back on, so if the service worker is healthy it serves the cached shell and the app's own offline indicator appears; if the cache is broken, the reload fails and the assertion catches it. The order is the recipe: cache, then sever, then reload.

Recipe 615 The Resource-Budget Guard

Problem: Page weight only ever grows. Every sprint adds one more font, one more tracking script, one more hero image nobody compressed, and no single change feels big enough to argue about, until the homepage is two megabytes and the mobile bounce rate has quietly doubled. Without a number in CI, "the site got heavy" is a feeling, not a failing build, and feelings do not block merges.

```
Scenario: Homepage stays under 500KB transfer
  Given I start recording network requests
  When I am on "/"
  Then the total transfer size should be less than 500000
```

How It Works: `I start recording network requests` is the built-in step that attaches Playwright response listeners and begins logging every request the page makes. Navigation then loads the homepage under that recorder. `the total transfer size should be less than 500000` is a custom step that sums each recorded response's `transferSize` (bytes over the wire, after compression) and asserts the total stays under your budget. Set the recorder before navigating, or the requests fire before anything is listening and the total reads zero.

At scale: A page-weight budget only bites if it is the same number across the portfolio, pull the limit from an env var (say `PERF_BUDGET_BYTES`) so a heavier marketing site and a lean app each get a sane ceiling without editing the scenario. Feed the recorded totals into your report metadata and you get a weight-over-time trend per project, which is the chart that finally wins the "do we really need a fourth analytics script?" conversation.

Recipe 616 The Third-Party-Free Smoke

Problem: Legal signed a contract: no tracking fires before the visitor consents. Then a marketing manager adds a tag through Google Tag Manager without telling engineering, and now the analytics beacon fires on page one for every EU visitor, a regulatory breach that lives in a third-party config nobody on your team can see in the codebase. The fine is real; the change that caused it never touched your repo.

```
Scenario: Critical pageload is third-party-free
  Given I start recording network requests
  When I am on "/"
  Then no request to "*googletagmanager.com*" should have been made
  And no request to "*facebook.net*" should have been made
  And no request to "*hotjar.com*" should have been made
```

How It Works: With the recorder running from the first step, `no request to "..." should have been made` is the built-in negative network assertion, it checks the recorded request log and fails if any URL matched the glob. The `*...*` pattern catches any host containing that domain, so it does not matter which subdomain or CDN edge the tracker loads from. Because it inspects actual outbound requests rather than your source, it catches trackers injected by tag managers, third-party widgets, or someone's "harmless" embed, the ones code review never sees.

At scale: Keep the blocklist of tracker domains in one shared JSON or env list and loop the assertion over it, so adding a newly-banned vendor is a one-line change that propagates to every consent-gated site at once. Tag these `@privacy` or `@consent`: when a regulator or a client asks "prove nothing tracks before consent," a tagged, green run across the portfolio is the evidence, far better than a screenshot and a promise.

Recipe 617 The Time-To-Interactive Guard

Problem: The page *looks* loaded, hero image, headline, the lot, but tap a button and nothing happens for two more seconds while the JavaScript is still parsing. Users read "looks ready" as "is ready," click, get no response, and click again, and your support inbox fills with "the button is broken." Visual completeness and interactivity are different moments, and the gap between them is where rage-clicks live.

```
Scenario: Homepage interactive in under 3 seconds
  Given I start the performance timer
  When I am on "/"
  And I wait until the page is interactive
  Then the elapsed time should be under 3000 milliseconds
```

How It Works: These three working steps are custom, a timer-start, an interactivity wait, and an elapsed-time assertion, because webship-js's philosophy is to wait for events, not clock time, so it ships no stopwatch of its own. The timer records a start mark; `wait until the page is interactive` resolves when the main thread is free enough to handle input (long tasks have

drained); the final step asserts the delta stayed under budget. Note this is deliberately the one place you measure wall-clock time on purpose, everywhere else, a fixed sleep is a smell. Here the elapsed number *is* the thing under test.

At scale: Performance gates are noisy on shared CI runners, where a busy host inflates timings and turns a real budget into a flaky one. Run these on a dedicated, consistently-sized runner, keep the budget in an env var per project, and treat a single breach as a warning, not an instant red, track the trend across runs instead. A performance number is only a portfolio signal when every project measures it on comparable hardware.

API Contract Patterns

Recipe 618 The Schema Match

Problem: The mobile team built against your `/api/users` response for three weeks. Then a backend dev renamed `full_name` to `name`, a one-word change, perfectly reasonable, and shipped it, and now every mobile client crashes on the field that vanished. The API had no test that said "this shape is a promise," so nothing stopped the rename, and the first sign of trouble was a spike in app-store one-star reviews.

```
Scenario: User endpoint matches schema
  When I send a GET request to "/api/users/1"
  Then the response should match JSON Schema "fixtures/user.schema.json"
```

How It Works: The GET runs on axios, no browser, milliseconds per call. `the response should match JSON Schema "..."` is a custom step that loads a JSON Schema file and validates the response body against it, failing with the exact path that drifted (`/full_name` missing, `/age` is a string not a number). A schema is stricter and more honest than asserting a handful of

fields by hand: it pins types, required keys, and structure in one artefact the API team can treat as the contract. Keep the schema in version control next to the test so a breaking change shows up as a failing diff, not a production incident.

Recipe 619 The Error-Shape Contract

Problem: Everyone tests the happy path. The 200 with the right body is well covered, but the 404, the 422, the 500 are an afterthought, and they are exactly what a client integrator hits first when something goes wrong. If your errors are inconsistent (`{error: "..."}` on one endpoint, `{message: "..."}` on the next, a raw stack trace on a third), every consumer writes bespoke error handling and your support load triples. The error shape is part of the API, even though nobody treats it that way.

```
Scenario: 404 error has the expected shape
  When I send a GET request to "/api/users/9999"
  Then the API response code should be 404
  And the JSON response should have property "error.code"
  And the JSON response should have property "error.message"
  And the JSON response should have property "error.docs_url"
```

How It Works: The request deliberately asks for a resource that does not exist, so the API takes its error branch. `the API response code should be 404` confirms the status, then `the JSON response should have property "error.code"` (a custom dotted-path existence check, sibling to the JSON-pointer assertions in the fact sheet) verifies each field the error contract promises, a machine-readable code, a human message, a link to the docs. Testing the error envelope, not just the status, is what makes the failure path as reliable as the success path.

At scale: Errors should be uniform across every service you run, or your clients pay the integration tax forever. Lift this into a Scenario Outline that hits each service's known-404 endpoint and asserts the same three fields, and the table becomes the org-wide error-shape contract. The first time a new service ships with a bare `{message}` instead of the standard envelope, this catches it before any consumer does.

Recipe 620 The Pagination Contract

Problem: Pagination is the API feature that breaks quietly. A refactor drops the `links.next` field, and every client that walks pages by following that link silently stops after page one, no error, no 500, just half the data quietly missing from a dashboard nobody scrolls past the first screen of. The bug surfaces a month later as "the report totals are wrong," and tracing it back to a missing JSON key takes an afternoon.

```
Scenario: Page 2 of articles
  When I send a GET request to "/api/articles?page=2&per_page=10"
  Then the API response code should be 200
  And the JSON response should have property "data"
  And the JSON response should have property "meta.total"
  And the JSON response should have property "links.next"
  And the JSON response should have property "links.prev"
```

How It Works: Asking for page 2 specifically (not page 1) forces both navigation links to exist, a `prev` that points back and a `next` that points forward, which is the case most likely to break. Each `should have property` is a dotted-path existence check against the response body: `data` carries the page of items, `meta.total` the count clients need to compute page counts, and the `links` pair the cursors they follow. You are pinning the envelope a paginated API depends on, the part that lets a consumer walk every page without guessing URLs.

Recipe 621 The Idempotent-POST Guard

Problem: A customer taps "Pay" once, the network stalls, they tap again, and your payment endpoint charges them twice. Idempotency keys exist to stop exactly this, but the logic is subtle, easy to break in a refactor, and catastrophic when it fails: double charges mean refunds, chargebacks, and a support thread that starts with a furious customer. This is the test you write before you ever ship money-moving code, and keep forever.

```
Scenario: Same idempotency key returns same resource
  When I send a POST request to "/api/payments" with body:
    """
    { "amount": 100, "idempotency_key": "abc-123" }
    """
  Then the API response code should be 201
  When I send a POST request to "/api/payments" with body:
    """
    { "amount": 100, "idempotency_key": "abc-123" }
    """
  Then the API response code should be 200
```

How It Works: The body docstrings carry the same `idempotency_key` on both POSTs. The first call should create the resource and return `201 Created`; the second, carrying an identical key, should be recognised as a replay and return `200 OK` pointing at the resource already made, not a second `201`, and crucially not a second charge. The status difference (`201` then `200`) is the observable proof the server de-duplicated. axios keeps the calls fast, and webship-js's API steps share state within the scenario so both requests run against the same backend in sequence.

At scale: Any endpoint that moves money or sends an irreversible action, payments, orders, emails, needs this guard, so standardise the key field name and the 201/200 convention across services and reuse the scenario shape everywhere. Tag these `@money` or `@critical`: a double-charge regression is the kind of failure where you want the whole portfolio's payment endpoints checked on every release, not just the service someone happened to touch.

Recipe 622 The Rate-Limit Header

Problem: A partner integrates against your API, gets no rate-limit headers back, and has no idea how close they are to the ceiling, so they hammer you until they hit a wall of 429s with no warning, then open a ticket blaming your API for "randomly failing." Well-behaved clients back off *if you tell them how much budget is left*. The headers are the contract that makes good behaviour possible, and a middleware reorder can strip them without anyone noticing.

```
Scenario: Response carries rate-limit headers
  When I send a GET request to "/api/articles"
  Then the response should contain the header "X-RateLimit-Limit"
  And the response should contain the header "X-RateLimit-Remaining"
  And the response should contain the header "X-RateLimit-Reset"
```

How It Works: A plain GET is enough, the headers ride on every response, no need to actually exhaust the limit. `the response should contain the header "..."` is a custom header-existence assertion that inspects the response headers axios captured: the ceiling (`Limit`), how many calls remain in the window (`Remaining`), and when the window resets (`Reset`). Checking presence here is the point; the values change every request, but a missing header means clients are flying blind, and that is the regression you are guarding against.

At scale: Rate-limiting is usually one shared gateway or middleware in front of many services, so when it regresses it regresses for all of them. Run this trio against one representative endpoint per service and you confirm the shared layer is intact everywhere with a handful of millisecond-cheap GETs. Standardise on one header convention (the `X-RateLimit-*` family) across the org so partners learn it once and the test reads identically in every repo.

Form Patterns

Recipe 623 The Required-Field Sweep

Problem: Someone adds a new field to the signup form, marks it required in the markup, and forgets to wire the validation message, so submitting it blank either fails with a generic error or, worse, lets the empty value through. Required-field validation is tedious to test by hand for every field, so it usually is not, and the gaps show up as malformed records in your database long after signup.

```
Scenario Outline: <field> is required
  Given I am on "/signup"
  When I leave "<field>" blank
  And I press "Sign up"
  Then I should see "required" near "<field>"

  Examples:
    | field    |
    | name     |
    | email    |
    | password |
```

How It Works: The Scenario Outline expands into one independent scenario per row, so a broken validation on `email` fails as its own named line in the report instead of hiding in a loop. `I leave "<field>" blank` and `I should see "required" near "<field>"` are custom steps, the first ensures the field is empty, the second scopes the “required” text to that field’s region so a generic error elsewhere does not pass it by mistake. `I press "Sign up"` is the built-in button press, which triggers `smartSettle` to let client-side validation render before the assertion polls.

At scale: The list of required fields lives in the Examples table, so adding a field to the form is one new row, not a new scenario, and a QA owning many signup forms reads the table as the spec for “what must be filled in here.” When several products share a signup component, keep the table

shape identical so the same outline drops into each suite and “did required-field validation break anywhere?” stays a one-query question.

Recipe 624 The Email-Validation Sweep

Problem: Email validation is where teams either do too little or get it subtly wrong. Accept `foo@` and you fill the database with addresses that bounce every newsletter; tighten the regex too far and you reject someone’s perfectly valid `name+tag@sub.domain` address and lose the signup. The only way to know which malformed inputs your form actually catches is to throw a table of them at it, and nobody does that by hand for every release.

```
Scenario Outline: Email "<value>" is invalid
  Given I am on "/signup"
  When I fill in "email" with "<value>"
  And I press "Sign up"
  Then I should see "valid email" near "email"

  Examples:
    | value          |
    | not-an-email   |
    | @example.com   |
    | foo@           |
    | foo bar        |
```

How It Works: Each row is a known-bad address that *should* be rejected; the outline runs them as separate scenarios so you see exactly which malformed value slipped through. `I fill in "email" with "<value>"` is the built-in label-first field step, it resolves the field by its label, then id/name/placeholder. After the button press triggers validation and `smartSettle` lets the message render, `I should see "valid email" near "email"` (a custom scoped-text step) confirms the error appeared next to the field. Keep a sibling scenario for the *valid* cases too, so a regex that rejects everything cannot pass this by being maximally strict.

Recipe 625 The Password-Strength Bar

Problem: The password-strength meter is a small bit of UX with an outsized job: it nudges users toward passwords that will not get their account taken over. When the scoring logic drifts, a library upgrade rescoring `password` as "strong," or the colour classes getting renamed in a redesign, the meter starts lying, telling users a weak password is fine. It is the kind of regression that ships unnoticed because the bar still *looks* like it works.

```
Scenario Outline: Password "<value>" has strength "<strength>"
  Given I am on "/signup"
  When I fill in "password" with "<value>"
  Then ".strength-bar" should have class "strength-<strength>"

  Examples:
    | value             | strength |
    | a                 | weak     |
    | password          | medium   |
    | P@ssw0rd!Strong   | strong   |
```

How It Works: The Examples table pins specific inputs to the strength class each should produce, a single character is `weak`, a common word is `medium`, a long mixed string is `strong`. `I fill in "password" with "<value>"` types into the field via the built-in label-first step, then `should have class "strength-<strength>"` polls the meter web-first (the strength class lands on the next render) and checks the expected class is present. The class-based assertion is robust to visual restyling, it tests the scoring decision, not the pixels, so a colour change does not break it but a scoring regression does.

Recipe 626 The CSRF Token Refresh

Problem: CSRF protection only works if the token actually rotates. If a caching layer freezes the token, a reverse proxy caching the form HTML, or a CDN rule that should have excluded the page, every visitor gets the same token, and the protection quietly becomes theatre. It is a security regression with zero visible symptoms: forms still submit, nothing errors, and only a penetration test (or an attacker) discovers the token never changes.

```
Scenario: CSRF token is fresh per request
  Given I am on "/posts/new"
  And I capture the CSRF token from the page
  When I wait for 1 second
  And I reload the page
  Then the CSRF token should be different from the captured value
```

How It Works: `I capture the CSRF token from the page` and `the CSRF token should be different from the captured value` are custom steps that read the token (from its hidden input or meta tag), stash it on the world, and later compare. The reload fetches a fresh form; a healthy backend issues a new token each time, so the values differ. The one-second wait is here only to make the two requests provably distinct in time, note webship-js's own guidance is to wait for events rather than the clock, so this fixed wait is a deliberate, narrow exception for a timing-sensitive security check, not a pattern to copy into ordinary flows.

Recipe 627 The Autosave Test

Problem: A writer spends an hour on a long post, the browser crashes, and the draft is gone, because the autosave that was supposed to fire every thirty seconds silently stopped working after a refactor. Autosave is the safety net nobody notices until it fails, and testing it the honest way would mean sitting and waiting thirty real seconds per run, which no one will do, so it goes untested until someone loses real work.

```
Scenario: Draft autosaves every 30 seconds
  Given the system time is "2026-01-01T00:00:00Z"
  And I am on "/posts/new"
  When I fill in the WYSIWYG field "Body" with the "Hello"
  And I advance the clock by 30 seconds
  Then ".autosave-status" should have text "Saved" within 1 second
```

How It Works: `Given the system time is "..."` installs Playwright's clock control via `page.clock`, freezing time at a fixed instant. `I fill in the WYSIWYG field "Body" with the "Hello"` is the built-in rich-text step that types into the editor's `contenteditable` body rather than a plain input. Then `I advance the clock by 30 seconds` fast-forwards the page's virtual clock, every `setTimeout` and `setInterval` waiting on that thirty-second mark fires instantly, so the autosave timer triggers without thirty real seconds passing. The status assertion then polls web-first until "Saved" appears. You test a half-minute behaviour in milliseconds.

At scale: Clock control turns every time-based feature, autosave, session timeout, polling refresh, scheduled banners, into a fast, deterministic test instead of a slow, flaky `wait`. Standardise on `Given the system time is "..."` at the top of these scenarios across products so the same fixed instant makes results reproducible, and tag them `@clock` so the time-dependent suite is easy to find and reason about when a timezone change breaks something.

Recipe 628 The File-Size Validation

Problem: The upload form is supposed to cap files at 5MB. Then the client-side check gets removed in a refactor, or the server limit and the message disagree, and a user drags in a 40MB video, which either uploads and fills your storage, or fails with a cryptic server error instead of the friendly "too big" message. File-size limits are exactly the kind of boundary that rots silently, because no one uploads a deliberately oversized file as part of normal use.

```
Scenario: Upload over 5MB is rejected
  Given I am on "/uploads/new"
  When I attach the file "fixtures/large-file-10mb.bin" to "file"
  And I press "Upload"
  Then I should see "exceeds the maximum size of 5 MB" near "file"
```

How It Works: `I attach the file "..." to "file"` is the built-in file-input step, it sets the chosen file on the `<input type="file">` without opening a native OS dialog, which is why upload tests are scriptable at all. The fixture is deliberately over the limit (a 10MB binary against a 5MB cap). After the upload press and `smartSettle`, `I should see "exceeds the maximum size of 5 MB" near "file"` (a custom scoped-text step) confirms the specific rejection message appeared by the field. Keep the oversized fixture in the repo so the boundary case is reproducible on every machine and in CI, not dependent on a file someone has to generate locally.

Search Patterns

Recipe 629 The Search-As-You-Type Test

Problem: Live search feels magical when it works and maddening when the debounce breaks. Lose the debounce and every keystroke fires a request, hammering the backend and flickering results; set it too long and the results lag behind the user's typing. The timing is the feature, and timing is the hardest thing to test reliably, a real `wait` makes the test slow and flaky, which is why search-as-you-type so often ships with no test at all.

```
  Scenario: Search updates after debounce
    Given I am on "/search"
    When I fill in "q" with "play"
    And I advance the clock by 300 ms
    Then ".search-result" should have a count of at least 1 within 2
seconds
```

How It Works: Typing into the query field starts the debounce timer the search uses to avoid firing on every keystroke. `I advance the clock by 300 ms` jumps the page's virtual clock past that debounce threshold (clock control must be installed for the page first), so the deferred fetch fires immediately instead of after a real 300ms wait. The count assertion then polls web-first up to two seconds for results to render. You are testing the debounce *and* the result render deterministically, the clock makes the timing exact rather than a hopeful sleep.

Recipe 630 The No-Results Test

Problem: Search for something that does not exist and the page goes blank, no message, no suggestion, just an empty void that looks broken. The empty state is the path developers never see, because they always search for things that *are* in their test data, so it ships unstyled or missing entirely and users assume the search itself is broken rather than that their query had no matches.

```
Scenario: No-results message shows
  Given I am on "/search"
  When I fill in "q" with "xyzzyqwertasdf"
  And I press "Search"
  Then ".no-results" should be visible within 2 seconds
```

How It Works: The query is deliberate gibberish chosen to match nothing, forcing the empty branch. After `I press "Search"` submits and `smartSettle` lets the response render, `should be visible within 2 seconds` polls the no-results element web-first (100ms intervals) until it appears or the budget runs out. The two-second window covers a real search round-trip. The whole point is asserting the empty state is a designed, visible message, not the absence of results, which a naive test would read as a silent pass.

Recipe 631 The Search-Suggestion Test

Problem: Autocomplete suggestions are what make a search box feel fast, and they break in ways that are easy to miss: the suggestion service times out, the endpoint moves, the dropdown renders empty. Users start typing, see nothing helpful, and type the whole query the slow way, or give up. Because suggestions appear and vanish quickly, a developer clicking through manually rarely notices when they have stopped showing.

```
Scenario: Suggestion appears for partial input
  Given I am on "/"
  When I fill in "q" with "play"
  Then ".suggestions li" should have a count of at least 3 within 1
second
```

How It Works: Typing a partial query triggers the suggestion fetch. `should have a count of at least 3 within 1 second` is the web-first count assertion, it polls the suggestion list every 100ms until at least three items exist or the second elapses, which neatly handles the async gap be-

tween typing and the dropdown populating without any manual wait. Asserting “at least 3” rather than an exact number keeps the test stable as the underlying content changes while still proving the suggestion machinery fired and returned useful results.

Recipe 632 The Highlight Test

Problem: Search highlighting, the bold or yellow mark on the matched term in each result, is a small touch that tells users *why* a result matched. When it breaks, results look arbitrary: the user searched “playwright,” sees a list, and cannot tell at a glance which word triggered each hit. It usually breaks when the highlighting moves server-side to client-side or a sanitiser strips the `<mark>` tags, and it is invisible unless you assert the markup is actually there.

```
Scenario: Match highlights in results
  Given I am on "/search?q=playwright"
  Then ".result mark" should contain text "playwright"
```

How It Works: Navigating straight to a results URL with the query in it skips the typing and lands on a rendered result set. `.result mark` targets the `<mark>` element search engines use to wrap matched terms inside each result, and `should contain text "playwright"` polls web-first until that highlighted element holds the search term. Testing for the `<mark>` element specifically proves the highlighting is semantic markup, the thing screen readers and styling both rely on, not just coincidental bold text that happens to look right.

Recipe 633 The Search-Result-Count Test

Problem: The header says “12 results” and the page shows 9. Users notice instantly, it reads as a bug even if both numbers are individually defensible, and it erodes trust in the whole search. The mismatch creeps in when the count comes from one query (a fast `COUNT(*)`) and the list from another (a paginated fetch), and the two drift apart after a caching or pagination change that touched only one of them.

```
Scenario: Result count matches actual matches
  Given I am on "/search?q=test"
  Then ".result-count" should have text "12 results"
  And ".result" should have a count of 12
```

How It Works: Two assertions cross-check each other. `should have text "12 results"` reads the count label the page renders, and `should have a count of 12` is the web-first DOM-counting assertion that tallies the actual `.result` elements present. When they agree, the displayed number is honest; when they disagree, you have caught the count-versus-list drift, the exact failure a single assertion would miss. Both poll web-first, so they wait out any async render before comparing.

At scale: The displayed-count-versus-actual-count check is a reusable shape for any “N items” UI, a cart total, a notification badge, a table footer, not just search. Lift it into a step or a tagged convention so the pattern travels across products, and the “header says one thing, list shows another” class of bug gets a standing guard everywhere it can occur instead of being rediscovered per project.

Navigation Patterns

Recipe 634 The Breadcrumb Test

Problem: Breadcrumbs are how users (and search engines) understand where a deep page sits in the hierarchy. When they break, a routing refactor reorders the trail, or a level drops out, navigation gets confusing and your structured-data SEO quietly degrades. Breadcrumbs are easy to overlook in testing because the page still works without them; you only feel the loss when someone four levels deep cannot find their way back up.

```
Scenario: Breadcrumb is correct
  Given I am on "/articles/category/playwright/42"
  Then the text "Articles" should appear after the text "Home"
  And the text "playwright" should appear after the text "Articles"
  And the text "Article 42" should appear after the text "playwright"
```

How It Works: `the text "X" should appear after the text "Y"` is the built-in ordering assertion, it checks document source order, confirming each crumb sits after its parent in the trail. Chaining the three assertions pins the full ancestry: Home, then Articles, then the category, then the leaf. Asserting *order* rather than mere presence is what makes this meaningful, a breadcrumb with all the right labels in the wrong sequence is still broken, and only the ordering check catches that.

Recipe 635 The Active-Menu-Item Test

Problem: The active-state highlight in the nav is the user's "you are here" marker. When it breaks, the active class lands on the wrong item, or on nothing, people lose their sense of place in the site, and on a deep section it is genuinely disorienting. It is fragile logic (matching the current path against each link) that quietly breaks on a routing change, and it looks fine on the homepage where most people glance, so it ships broken on the inner pages nobody re-checks.

```
Scenario: Active menu item highlights
  Given I am on "/articles"
  Then "nav a[href='/articles']" should have class "active"
  And "nav a[href='/']" should not have class "active"
```

How It Works: Two complementary web-first assertions make this airtight. `should have class "active"` confirms the link matching the current path is highlighted; `should not have class "active"` confirms the *home* link is not, guarding against the classic bug where the active state sticks to multiple items or to the wrong one. Testing both the positive and the negative is the trick: a highlight that turns *everything* active would pass a presence-only check but fails the negative assertion here.

Recipe 636 The Skip-Link Test

Problem: A keyboard user lands on your page and has to Tab through forty navigation links before reaching the actual content, on every single page. The skip link exists to spare them that, and it is the very first thing a screen-reader user encounters, but because mouse users never see it, it is the easiest accessibility feature to break in a header refactor and the last anyone notices. An accessibility audit flags it; by then it has shipped to every page on the site.

```
Scenario: Skip link reaches main
  Given I am on "/"
  When I press the key "Tab"
  Then "a.skip-link" should be focused
  When I press the key "Enter"
  Then "#main" should be focused within 1 second
```

How It Works: This walks the exact keyboard journey. `I press the key "Tab"` (built-in keyboard step) moves focus to the first focusable element, which on an accessible page is the skip link, `should be focused` is the web-first focus assertion confirming it. `I press the key`

`"Enter"` activates it, and the second focus assertion confirms focus actually moved into `#main`, polling web-first because the focus shift can be async. Both halves matter: a skip link that is focusable but does not move focus on Enter is broken, and only checking the destination catches that.

At scale: The skip link is shared-header markup, so this is a once-per-project guard that protects every page, and it belongs in the same `@a11y` lane as your axe-core audits (Recipe 642) since axe checks structure but not the keyboard *behaviour* this verifies. Keep the `a.skip-link` and `#main` selectors consistent across products and the same scenario drops into any suite unchanged, accessibility regressions are exactly the ones you want caught uniformly across a portfolio, not project by project.

Recipe 637 The Back-Navigation Test

Problem: A user scrolls deep into a listing, taps into an article, reads it, hits Back, and lands at the top of the list instead of where they were, or on a blank page, or somewhere else entirely. Broken back-button behaviour is one of the most common and most infuriating web bugs, especially once a site adds client-side routing or aggressive caching, and it is rarely tested because the happy-path forward navigation is the only thing anyone clicks through.

```
Scenario: Back returns to listing
  Given I am on "/articles"
  When I click "Read more" in the "First Article" row
  Then the path should be "/articles/1"
  When I move backward one page
  Then the path should be "/articles"
```

How It Works: `I click "Read more" in the "First Article" row` is the built-in row-scoped click, it finds the row whose text matches “First Article” and clicks the “Read more” within it, so the test does not depend on the link’s position. `the path should be "/articles/1"` confirms the forward navigation, then `I move backward one page` is the built-in history-back step (Playwright’s `page.goBack()`), and the final assertion confirms the browser returned to the listing path. `smartSettle` runs after each navigation so both path checks read a settled URL, not one mid-transition.

Internationalization Patterns

Recipe 638 The Language-Switch Test

Problem: The site shipped in seven languages, and the language switcher is the one control every non-English user touches first. It breaks in the worst way, the click lands, the label changes, but the URL never switches locale, so the visitor reads English with an Arabic flag in the corner. Nobody on an English-speaking team notices, because nobody on the team clicks the switcher. The first report comes from a customer who assumed the Arabic site simply did not exist.

```
Scenario: Switch to Arabic
  Given I am on "/en"
  When I click "العربية"
  Then the path should match "^/ar"
  And the page language should be "ar"
```

How It Works: `When I click "العربية"` resolves the switcher by its visible text, the native language name, which is exactly what a user sees and clicks, so the test reads like the user's own action. The click triggers navigation, and `smartSettle` holds until the new locale page has loaded and gone quiet before either assertion runs. `the path should match "^/ar"` polls web-first against the URL until it starts with the Arabic locale prefix; the second assertion confirms the document's own `lang` attribute followed, which is the part screen readers and search engines actually read. Asserting both matters: a switcher that changes the URL but not the `<html lang>` is half-broken, and only checking both catches it.

At scale: Drive this with a Scenario Outline over your full locale matrix and the native label for each, one table row per language, the same scenario body. When several multilingual products share that table shape, "did the German switch break anywhere?" is a single tag query across reports rather than a manual click-through of every site.

Recipe 639 The RTL Layout Test

Problem: Right-to-left is the regression that hides behind a passing suite. The Arabic page loads, the translations are correct, every functional test is green, and the layout is mirrored wrong, sidebar on the left where it should be on the right, the whole reading order fighting the language. It usually breaks when a developer adds a component with hard-coded `left:` and `right:` properties instead of logical ones, and it lands in production because the only people who would catch it by eye are the people who do not review the Arabic build.

```
Scenario: Arabic page is RTL
  Given I am on "/ar"
  Then "html" should have attribute "dir" with value "rtl"
  And "body" should have class "rtl"
```

How It Works: Both lines are web-first attribute and class assertions, they poll the element every 100ms for up to five seconds until the value matches, so a `dir` attribute set late by client-side locale detection still passes without a hand-written wait. They check the two switches the rest of the layout hangs off: `<html dir="rtl">` is what flips the browser's own bidirectional engine, and the `rtl` class on `body` is the hook most CSS frameworks use to swap directional rules. This recipe asserts the *contract*, the flags are set, not every mirrored pixel. The flags being right is what makes the CSS work, and it is the cheap, stable thing to guard.

At scale: Run the same two-line check on every RTL locale across every multilingual project, keyed to a shared `@rtl` tag. The directional flags are the same regardless of which framework a given site uses, so one convention covers the whole portfolio, and when a component library ships a regression that drops the `dir` attribute, every RTL suite fails on the same line at the same time, which points straight at the shared dependency.

Recipe 640 The Currency-Formatting Test

Problem: A German customer sees their total as `1,234.56 €` instead of `1.234,56 €` and reads it as one euro and change instead of over a thousand. Currency formatting is locale-specific in ways English-speaking developers forget, the decimal and thousands separators swap, the symbol moves sides, and the bug is invisible in the codebase because the number is correct; only its presentation is wrong. It breaks when someone formats with a hard-coded `toFixed(2)` and a literal comma instead of the locale-aware formatter.

```
Scenario Outline: Currency formats by locale
  Given I am on "/<locale>/prices"
  Then I should see "<expected>"

  Examples:
    | locale | expected      |
    | en-US  | $1,234.56     |
    | de-DE  | 1.234,56 €    |
    | ar-SA  | 1,234.56 ر.س. |
```

How It Works: The Scenario Outline expands to one independent scenario per locale, so a broken German format fails as its own named line in the report, `Currency formats by locale (de-DE)`, instead of hiding inside a single multi-locale check. Each scenario navigates to that locale's price page and asserts the exact formatted string is present on the page. The separators are the whole point: `en-US` uses comma-thousands and dot-decimal, `de-DE` reverses both and moves the symbol to the end, `ar-SA` carries the Arabic currency symbol. Pinning the literal expected string is what catches a formatter that silently falls back to the default locale.

At scale: Keep one canonical price, here `1234.56`, across every locale and every project so the expected column is the only thing that changes per row. The day a shared formatting utility regresses, the failing rows across all your suites point at the same value formatted wrong, and you debug the library once instead of per product.

Recipe 641 The Date-Locale Test

Problem: `05/04/2026` means May 4th to an American and April 5th to everyone else, and that ambiguity has cancelled flights and missed deadlines. Date formatting is the most quietly dangerous locale bug because the wrong format is still a valid date, nothing errors, nothing looks broken, the reader just understands the wrong day. And you cannot test it reliably against the real clock, because "two days ago" changes every time the suite runs.

```
Scenario Outline: Date formats by locale
  Given the system time is "2026-05-14T00:00:00Z"
  And I am on "/<locale>/articles/1"
  Then ".published-date" should have text "<expected>"

  Examples:
    | locale | expected         |
    | en-US  | May 14, 2026     |
    | en-GB  | 14 May 2026      |
    | de-DE  | 14. Mai 2026     |
    | ar     | 14 مايو 2026     |
```

How It Works: `Given the system time is "..."` installs `page.clock` and pins the browser's notion of "now" to a fixed instant before navigation, so any relative date logic on the page computes against May 14th every run, the date assertion becomes deterministic instead of drifting with the calendar. Each outline row then loads that locale's article and checks the rendered date string exactly: month name and order differ per locale (`May 14, 2026` vs `14 May 2026` vs `14. Mai 2026` vs the Arabic month name). Freezing the clock is what makes a date test trustworthy in CI, without it, the expected column would be wrong tomorrow.

At scale: The frozen timestamp is your shared fixture; reuse the same instant across every date test in every project so a failure always means a formatting change, never a calendar change. Pair it with the currency outline from the previous recipe under one `@i18n` tag, and a locale rollout becomes a single suite you can point at any site to answer "does this product format dates correctly for all our markets?"

Accessibility Patterns

Recipe 642 The AA Audit Sweep

Problem: The accessibility audit happens once, before launch, a consultant signs off, and then the site evolves for two years with nobody re-running it. Then a procurement officer asks for a current VPAT, or a regulator does, and you discover that a hundred commits of new components quietly reintroduced the contrast and label failures the original audit cleared. Accessibility is not a milestone you pass once; it is a property that decays with every merge unless something guards it on every run.

```
Scenario Outline: <page> passes WCAG AA
  Given I am on "<page>"
  Then the page should pass an accessibility audit at level "AA"

  Examples:
    | page         |
    | /            |
    | /articles    |
    | /search      |
    | /contact     |
```

How It Works: `the page should pass an accessibility audit at level "AA"` runs axe-core against the live, fully-rendered DOM, because the step runs after `smartSettle`, it audits the page in its settled post-JavaScript state, not the initial HTML, which is where dynamically-inserted accessibility bugs live. axe checks the WCAG 2.x A and AA rules and the step fails on any violation at that level, naming the rule and the offending element. The Scenario Outline turns one audit into one named result per page, so a regression on `/search` fails as its own line instead of a single opaque “a11y failed”. This is automated coverage of the machine-detectable subset of WCAG, roughly a third of the criteria, not a substitute for manual testing, but it never forgets and never gets skipped under deadline.

At scale: Keep the page list short and high-traffic per project, the routes that matter, not every URL, and standardize the `AA` level across the portfolio so “are we still accessible?” has one answer shape everywhere. Feed the per-page results into your report metadata, and a delivery manager gets a comparable accessibility signal across twenty products without commissioning twenty audits.

Pro Tip: Audits run on what is rendered, so seed the page into a realistic state first, log in, expand the menu, open the cookie banner, or you only test the empty shell. The violations users hit are usually behind the first interaction.

Recipe 643 The Critical-Only Audit

Problem: You inherit a legacy product with three hundred minor accessibility violations, and a full AA gate would be red on day one and stay red for a year, which means everyone learns to ignore it, which means it guards nothing. But the checkout still has to work for a screen-reader user, or you are leaving revenue and a lawsuit on the table. You need a gate that is achievable today and still blocks the violations that actually stop people from buying.

```
Scenario: No critical accessibility violations
  Given I am on "/checkout"
  Then the page should have no critical accessibility violations
```

How It Works: This runs the same axe-core engine as the full audit but fails only on violations axe rates `critical`, the impact tier reserved for problems that outright block a user, like a form control with no accessible name or a trap that keyboard focus cannot escape. Moderate and minor findings are reported but do not fail the step. That makes it a realistic gate for an imperfect codebase: it holds the line on the failures that break the task while you burn down the long tail separately. On a checkout, “critical” maps almost exactly to “a disabled user cannot complete the purchase.”

At scale: Use critical-only as the enforced gate on every project’s revenue-path and sign-up pages, and run the full AA sweep (previous recipe) as a non-blocking report on the side. The two-tier split, block on critical, track on the rest, lets a portfolio with wildly different legacy debt share one

accessibility policy without grinding any single team to a halt.

Recipe 644 The Heading-Hierarchy Test

Problem: A screen-reader user navigates a long article by jumping heading to heading, it is how they skim. When a developer reaches for an `<h4>` because it "looked the right size" and skips `<h2>` and `<h3>`, that navigation breaks: the outline reads as a document with chapters but no sections, and the reader loses the thread. Sighted reviewers never catch it because the page looks fine; the headings are styled, just structured wrong.

```
Scenario: Heading structure is valid
  Given I am on "/articles/42"
  Then the heading hierarchy should be valid
  And the page should have exactly one h1
```

How It Works: `the heading hierarchy should be valid` is a structural check that walks every `<h1>`-`<h6>` in document order and fails if a level is skipped on the way down, an `<h2>` followed directly by an `<h4>` is the classic failure. `the page should have exactly one h1` enforces the single top-level heading that defines the page's title for assistive tech; zero h1s or several are both common CMS mistakes. Together they assert the document forms a clean tree, which is precisely what heading-based navigation depends on. This is a check styling cannot reveal, it reads the tag levels, not the rendered sizes.

At scale: Headings are the cheapest structural signal to standardize across a content portfolio, because the rule is identical for every page on every site, one level at a time, exactly one h1. Run it on a sample of each content type per project; a CMS template that emits a skipped level will fail the same way everywhere, pointing at the shared template rather than the content.

Recipe 645 The Image-Alt Sweep

Problem: The gallery is the page most likely to fail this and the page nobody checks. Editors upload images through the CMS for months, and somewhere in there the “alt text” field gets left blank, because it is optional in the UI, and deadlines are not. A blind user arrives at a wall of “image, image, image, image” with no idea what any of them show. It is the single most common accessibility violation on the web, and it accrues one forgotten upload at a time.

```
Scenario: All images have alt text
  Given I am on "/gallery"
  Then every image should have an alt attribute
```

How It Works: The step selects every `<img>` in the settled DOM and fails if any one is missing the `alt` attribute, naming the offenders so you know which images to fix. Worth knowing: a present-but-empty `alt=""` is valid and intentional for purely decorative images, it tells a screen reader to skip them, so the check is for the attribute’s *presence*, not for non-empty text. The hard part of alt text (is it *good* text?) stays human; this guards the floor, which is that the attribute exists at all. On a content-heavy page that floor is exactly where the failures cluster.

At scale: Run this against each content type’s representative page across the portfolio, and pair it with an editor-side reminder, because the real fix is upstream in the CMS workflow, not in the test. The sweep is your safety net: when a content team’s discipline slips on one project, the gallery scenario goes red before the missing alt text ships.

Recipe 646 The Form-Label Sweep

Problem: A designer replaces visible labels with placeholder text because it looks cleaner, “Email” sitting greyed inside the box instead of above it. It demos beautifully. Then a screen-reader user tabs into the field and hears “edit text, blank,” because a placeholder is not a label. Sign-up is the worst place for this to happen: it is the first form a new user meets, and a field they cannot identify is a field they cannot fill.

```
Scenario: All form fields have labels
  Given I am on "/signup"
  Then every form field should have an accessible label
```

How It Works: This walks every form control and checks each one has an *accessible name* by any of the valid means, a `<label for>` association, `aria-label`, `aria-labelledby`, or a wrapping label, the same name-computation a screen reader performs. It deliberately does not accept `placeholder` as a label, which is exactly the trap this catches: a placeholder-only field passes a casual eyeball and fails this step. A field with no accessible name is a field assistive tech announces as anonymous, and on a sign-up form that is a hard stop.

At scale: Label resolution is identical across stacks, so this one scenario shape covers the sign-up, contact, and search forms on every project unchanged. Track which forms it runs against per product, and a single shared design-system regression, a new input component that drops its label association, lights up the same failure across every suite at once.

Recipe 647 The Focus-Ring Test

Problem: Early in a project, someone hits `outline: none` to kill the "ugly" blue ring around buttons, and never puts a replacement back. The design looks crisp. Then a keyboard-only user tabs through the page and has no idea where they are, because the one visual cue that tracks focus is gone. It is the most-deleted accessibility feature on the web, removed for aesthetics by people who navigate with a mouse and never feel the loss.

```
Scenario: Focus ring is visible on Tab
  Given I am on "/"
  When I press the key "Tab"
  Then the focused element should have visible focus indicator
```

How It Works: `When I press the key "Tab"` moves focus to the first focusable element exactly as a keyboard user would, and `smartSettle` lets any focus-driven style change land before the assertion. `visible focus indicator` is a custom step that reads the focused element's

computed style and asserts it carries a non-default `outline` or `box-shadow`, in other words, that the browser's default ring was either kept or deliberately replaced, not stripped to nothing. This catches the specific regression a visibility check never could: an element that is perfectly visible but gives no signal that it holds focus.

At scale: Codify this custom step once in your shared step package (the cross-project sharing pattern later in this chapter) so every suite measures "visible focus" the same way, otherwise each team invents its own threshold and the results stop being comparable. A single design-token change that zeroes out the focus style then fails the same assertion across every project that imports the package.

Robustness Patterns

Recipe 648 The Stale-Element Guard

Problem: This is the flake that gets a scenario quarantined for no good reason. A dashboard widget refreshes itself, polls the server, re-renders, and a test that grabbed a button handle a moment earlier clicks a node that no longer exists in the DOM. In a naive framework that throws a "stale element reference" and the test fails maybe one run in ten, always on a different line, which is exactly the profile that makes a team give up and add a retry instead of finding the cause.

```
Scenario: Action survives DOM mutation
  Given I am on "/dashboard"
  When I refresh the widget "stats"
  And I click "View details" in the "stats" widget
  Then "/stats" should be the current path
```

How It Works: The widget refresh re-renders that part of the DOM, replacing the old nodes. The defence is that webship-js never holds a node reference across steps, every action re-resolves its locator at the moment it runs, against the live DOM. So `I click "View details"` looks up the

button fresh, after the refresh, and clicks whatever node currently matches, there is no stale handle to go bad. On top of that, the AfterStep `smartSettle` lets the re-render's mutations and any in-flight fetch settle before the click resolves. The recipe proves a stable property: an action that fires *after* a mutation lands on the new element, not a ghost of the old one.

At scale: Self-refreshing widgets, live stats, notification bells, ticker tapes, are everywhere in modern dashboards, and they are the number-one source of "works alone, flakes in the suite" reports. Make this guard a template across your dashboard-heavy projects so the re-resolution behaviour is tested, not assumed, and so a genuinely stale-prone custom step gets caught here instead of in a 2 a.m. CI run.

Recipe 649 The Race-Condition Guard

Problem: An impatient user double-clicks "Save", everyone does it, on a slow connection especially, and your backend gets two identical POSTs. If the submit button does not disable itself on first click, that is a duplicate order, a double charge, two copies of the same record. It is the bug that does not show up in any single-click test, only in the wild, where real users click the way real users click. The day it bites is the day finance asks why a customer was billed twice.

```
Scenario: Two rapid clicks fire only one request
  Given I start recording network requests
  And I am on "/"
  When I click "Save" twice in rapid succession
  Then exactly 1 POST request to "/api/save" should have been made
```

How It Works: `Given I start recording network requests` arms webship-js's request recorder, which logs every request the page makes for the rest of the scenario. `click "Save" twice in rapid succession` is a custom step that fires two clicks within 50ms, deliberately faster than a human but exactly what a fast double-click looks like, and faster than most debounce intervals. The assertion then counts the recorded POSTs to `/api/save` and demands exactly one: proof the UI swallowed the second click (by disabling the button, setting an in-flight flag, or de-

bouncing the handler) before it became a second request. Asserting against the recorded network traffic, not the UI state, is what makes this airtight, you are checking what actually left the browser.

At scale: Double-submit protection is a property every form with a side effect needs and almost none get tested for. Standardize the custom step and the recording pattern across projects so any checkout, any "create" form, any irreversible action carries the same guard. One shared step, audited once, beats discovering per project which forms forgot to disable on submit.

Recipe 650 The Slow-Render Guard

Problem: One slow third-party widget, a recommendations panel, a social feed, an ads call, holds the entire homepage hostage because someone awaited it in the critical render path. On a fast office connection nobody notices. On a real mobile connection in a tunnel, the whole page sits blank for five seconds while one non-essential box dawdles. The architecture sin is invisible until the network is slow, which is to say invisible to everyone testing on the office wifi.

```
Scenario: Slow widget doesn't block first paint
  Given the URL "/api/slow-widget" is delayed by 5000 ms
  When I am on "/"
  Then "h1" should be visible within 2 seconds
```

How It Works: `Given the URL "/api/slow-widget" is delayed by 5000 ms` installs a `page.route()` handler that holds responses to the widget endpoint for five seconds before fulfilling them, a deterministic slow dependency you control, no real network throttling needed. The assertion then demands the `h1` is visible within two seconds, a window that closes well before the widget can possibly arrive. If the page architecture renders the shell independently and hydrates the widget asynchronously, the header paints fast and the test passes; if the main content is gated behind that one slow call, the two-second budget expires and it fails. The deliberate gap between the 5s delay and the 2s budget is the whole design, it forces the answer.

At scale: Pick the one genuinely-slow-able dependency on each critical page and guard its independence. Across a portfolio this is how you stop "fast on staging, slow in production" surprises, the failure tells you precisely which project coupled its first paint to a dependency it should have deferred, before a real user on a real phone finds out.

Recipe 651 The Error-Boundary Test

Problem: A backend hiccups and returns a 500 for one widget. The question is whether your dashboard degrades gracefully or detonates. The bad outcome has two faces: a single failed widget takes down the whole page with a white screen of death, or, nearly as bad, the widget shows the user a raw `TypeError: Cannot read property 'data' of undefined`, leaking your stack trace and terrifying a non-technical user. A real failure should land as a calm "couldn't load," scoped to the one box that broke.

```
Scenario: Component error renders boundary
  Given the URL "/api/widget" returns status 500
  When I am on "/dashboard"
  Then ".widget" should have class "error" within 5 seconds
  And ".widget" should contain text "couldn't load"
  But ".widget" should not contain text "TypeError"
```

How It Works: The mock forces the widget's data endpoint to return a 500, simulating a real backend failure deterministically. The three assertions then test the full contract of a good error boundary. `should have class "error"` confirms the boundary actually caught the failure and entered its error state. `should contain text "couldn't load"` confirms the user gets a human message instead of a blank box. And the `But ... should not contain text "TypeError"` is the assertion most teams forget: it proves the raw error did not leak into the UI, no stack trace, no internal symbol names, nothing that exposes the implementation or alarms the reader. Each is web-first, polling up to its budget while the boundary renders. The boundary catches the error, the user sees a friendly message, and the raw error stays hidden.

At scale: Every component that fetches its own data needs a boundary, and "no raw error text leaks to the user" is a portfolio-wide security and trust rule, not a per-widget nicety. Standardize the negative assertion, never show `TypeError`, `undefined`, a stack frame, across projects, and you catch the day a refactor strips a boundary and starts spilling internals into the page.

Test-Suite Hygiene Patterns

Recipe 652 The Flaky-Quarantine

Problem: One scenario fails maybe one run in twenty, never the same way twice, and you cannot fix it today. So the team starts hitting "re-run", and within a month a red build means nothing, because everyone assumes it is "just the flaky one." That is the real damage of a flaky test: not the test, but the trust it bleeds out of the whole suite. A green build has to mean green, or the suite stops being a gate at all.

```
@flaky
Scenario: Known flaky scenario
  ...
```

How It Works: The `@flaky` tag splits one suite into two lanes by tag expression. The trustworthy lane runs `cucumber-js --tags "not @flaky"`, it excludes every quarantined scenario, so a failure there is real and blocks the merge. The quarantine lane runs `--tags @flaky --retry 3`, giving the unstable ones up to three attempts and reporting their outcome without gating anything. The point is honesty: the gate stays trustworthy, the flaky tests still run and still produce data, and nobody learns to ignore red. The discipline that makes it work is the audit, review `@flaky` weekly and either fix the test or delete it, because a quarantine with no exit becomes a graveyard of scenarios nobody trusts and nobody removes.

At scale: A flaky-rate-per-project metric, pulled from how many scenarios carry `@flaky` in each repo, is one of the sharpest health signals a QA lead owns, a suite drifting toward 10% flaky is a suite about to lose its team's trust. Standardize the tag and the two-lane CI split everywhere so the number is comparable, and make "drive `@flaky` back to zero" a tracked goal rather than a someday wish.

Recipe 653 The WIP Tag

Problem: A developer commits a half-written scenario for a feature still in progress, the steps exist, the feature does not, and the whole team's pipeline goes red on something that was never supposed to be tested yet. Now everyone is blocked on one person's unfinished work, and the Slack thread starts. Work-in-progress is normal and necessary; it just has no business breaking the shared build.

```
@wip
Scenario: In progress
  ...
```

How It Works: The `@wip` tag carves out a private lane. The default CI profile filters it out, `--tags "not @wip"`, so unfinished scenarios never run in the shared pipeline and can never turn it red. The author runs them locally, by tag, while building the feature. When the feature ships, the tag comes off and the scenario joins the real suite in the same commit. The effect is exactly the inverse of the flaky tag: `@flaky` runs everywhere but gates nowhere, `@wip` gates nowhere and runs only locally. Both keep the shared build honest by quarantining a different kind of "not ready."

At scale: Add `not @wip` to the default tag filter in every project's shared CI config so the convention is automatic, not something each developer has to remember to honour. Then a `@wip` scenario is safe to commit and push in any repo from day one, which is what makes developers willing to commit early instead of hoarding work on a branch.

Recipe 654 The Scenario-Per-Bug

Problem: A bug gets fixed, the ticket closes, everyone moves on, and four months later a refactor quietly reintroduces it, and you fix the exact same bug a second time. Without a test pinning it shut, every closed bug is only closed until someone unknowingly reopens it. The most expensive bugs are the ones you pay to fix twice, and they are entirely preventable with one scenario written at the moment of the fix.

```
@regression @bug-1284
Scenario: Empty coupon does not crash
  Given I am on "/checkout"
  When I click "Apply" without entering a coupon
  Then I should see "Enter a coupon code"
  And there should be no JavaScript errors
```

How It Works: This scenario reproduces the exact path that broke, empty coupon, click Apply, and asserts the now-correct behaviour: a polite validation message instead of a crash. The two tags do real work. `@regression` puts it in the slower, comprehensive lane that catches drift over time. `@bug-1284` ties the scenario to the ticket, so when it ever fails again the message points straight at the original bug, the fix, and the discussion, instant context, no archaeology. The closing `And there should be no JavaScript errors` reads the scenario's JS error buffer (the page-error and console listeners webship-js installs) and fails if the empty-coupon path threw anything, because the original symptom here was an uncaught error, and a "fix" that hides the message but still throws is no fix at all.

At scale: Make scenario-per-bug a step in the bug-closing checklist on every project, with the `@bug-<id>` convention shared across repos. The regression suite then becomes a living, executable changelog of every defect the portfolio ever shipped, and the count of `@bug-*` scenarios per project is a quiet measure of how much hard-won knowledge each suite has captured rather than re-learned.

Recipe 655 The Smoke-as-Doc Convention

Problem: A new engineer joins and asks the question every new engineer asks: “what does this site actually do?” The wiki is eighteen months stale, the original PM has left, and the only fully current description of the product is the code, which is not something you can hand a newcomer over coffee. So onboarding becomes a fortnight of asking around, and the answers conflict, because everyone remembers a different version of the product.

How It Works: Treat the `@smoke` tag as living documentation. If your smoke scenarios are named in plain language, “User can log in,” “Checkout completes,” “Search returns results”, then the list of smoke scenarios *is* the answer to “what does this site do,” and it cannot go stale, because it has to pass on every run to stay green.

```
cucumber-js --tags @smoke --dry-run
```

`--dry-run` prints every smoke scenario name without executing a single one, Cucumber resolves and lists them, no browser, instant. The result is a live, code-backed product overview that is current by construction: a feature that no longer works fails its smoke scenario, so a smoke list that is all-green is a feature list that is all-true. That is documentation you can trust precisely because it is enforced.

At scale: When every project tags its smoke scenarios with the same plain-language discipline, `--tags @smoke --dry-run` becomes a portfolio catalogue, run it across repos and you have a current, accurate inventory of what every product does, generated from the suites themselves. For a delivery manager onboarding onto an unfamiliar product, that is the fastest honest overview available anywhere.

Recipe 656 The “One Failure, One Cause” Test

Problem: A scenario fails in CI. You re-run it on its own and it passes. The temptation is overwhelming: tag it `@flaky`, move on, ship. But a test that fails in the suite and passes alone is almost never flaky, it is telling you something true and uncomfortable, that another scenario left state behind that broke this one. Bury that signal under a retry and you have hidden a real ordering bug that will resurface as a much stranger failure later, when the suite shuffles.

How It Works: Before you reach for the flaky tag, run the failing scenario in complete isolation:

```
cucumber-js path/to/feature.feature --name "Scenario name"
```

The `--name` filter runs only the scenario whose name matches, nothing before it, nothing after. Now the diagnosis is binary and clean. If it *fails* alone, the bug is in the scenario itself, fix it. If it *passes* alone but *fails* in the full suite, the scenario is innocent; the fault is inter-scenario contamination, an earlier scenario created a user, left a cookie, seeded a record, mutated shared data the later one assumed clean. That is a real test-isolation bug, and the fix is making scenarios independent (per-scenario seeding, fresh contexts, cleanup hooks), not papering over it with `--retry`. Investigate before you retry; the retry only hides which scenario is the culprit.

At scale: Independence is the property that lets sharding and parallelism work at all, a suite full of order-dependencies cannot be split across the parallel jobs in the CI patterns later in this chapter. Make “passes in isolation” the standing bar across projects: every scenario must stand alone, and “it’s just flaky” must survive this two-command check before anyone is allowed to write it down.

CI Patterns

Recipe 657 The Sharded Run

Problem: The suite grew honestly, a scenario here, a regression there, and now it takes forty minutes, and a forty-minute pipeline is a pipeline developers route around. They stop waiting for it, merge on a guess, and the gate you built to catch problems becomes a thing people skip to get their work done. A slow suite does not just cost wall-clock time; it costs the discipline of waiting for green.

```
strategy:
  matrix:
    shard: [1, 2, 3, 4]
steps:
  - run: cucumber-js --shard ${{ matrix.shard }}/4
```

How It Works: `--shard 1/4` tells Cucumber to run only its quarter of the scenarios. The CI matrix spins up four jobs in parallel, each running a different shard, so all four quarters execute at once and wall time drops to roughly a quarter, forty minutes becomes ten. The one prerequisite is the property from the previous recipe: sharding distributes scenarios across machines arbitrarily, so any order-dependency between scenarios will fail unpredictably depending on how the split lands. Independent scenarios are what make horizontal scaling possible at all; this is where that discipline pays off.

At scale: Sharding trades money for time, four jobs cost four runners, so tune the shard count per project to its suite size rather than maxing it everywhere; a fast suite gains nothing from being split and just burns runners. The portfolio-level move is the same convention across repos so any team can dial shards up as its suite grows, without reinventing the matrix each time.

Recipe 658 The Browser-Matrix Run

Problem: Everything works in Chrome, because everyone develops in Chrome, and then a Safari user reports the date picker is dead, or a Firefox user cannot submit the form. The cross-browser bug is the one your team is structurally blind to, because the team's daily browser is the one browser that already works. You cannot afford to run every scenario in every engine on every push, but you cannot afford to find out about WebKit from a customer either.

```
strategy:
  matrix:
    browser: [chromium, firefox, webkit]
steps:
  - run: BROWSER=${{ matrix.browser }} cucumber-js --tags @cross-browser
```

How It Works: The `BROWSER` env var selects which engine `playwright.config.ts` launches, `chromium`, `firefox`, or `webkit` (the open-source engine behind Safari, which is how you test Safari behaviour without owning a Mac in CI). The matrix runs all three in parallel, each picking up its browser from the env var. The `--tags @cross-browser` filter is the cost control: only scenarios you have deliberately marked as browser-sensitive run three times. Everything else runs once, on the default browser, because most logic produces identical results across engines and paying triple for it is waste. You spend the triple cost precisely where engine differences actually bite, date inputs, CSS layout, media, focus handling.

At scale: Curating the `@cross-browser` tag is the real skill here, and it is worth doing once at the portfolio level: a shared checklist of what genuinely differs across engines, applied consistently, so no project either under-tests Safari or wastefully triple-runs its entire suite. That shared judgement is what keeps the cross-browser bill sane across twenty repos.

Recipe 659 The Always-Upload-Reports Step

Problem: A nightly run fails, you open CI the next morning to see why, and there is nothing, no report, no screenshots, just a red X and a wall of console log. So you re-run it locally hoping to reproduce, which you often cannot, because the failure was environment-specific. The evidence existed for exactly the moment you needed it and then evaporated, because the upload step ran only on success. The one run whose artifacts you actually need is the one run that never kept them.

```
- uses: actions/upload-artifact@v4
  if: always()
  with:
    name: cucumber-report
    path: tests/reports/
```

How It Works: By default a CI step is skipped once an earlier step in the job fails, which means a normal upload step never runs when the tests fail, exactly backwards from what you want. `if: always()` overrides that, forcing the upload to run regardless of the test outcome, so the cucumber JSON, the generated HTML report, and the failure screenshots survive the red build and land as a downloadable artifact. The failure case is when the report is most valuable and, without this one line, the case that loses it. It is a one-line change that converts “it failed, no idea why” into “it failed, here is the report and the screenshot of the exact step.”

At scale: Make `if: always()` non-negotiable in every project’s CI template, alongside webship-js’s failure screenshots (`WEBSHIP_SCREENSHOT_ON_FAILED` is on by default) and on-failure video (`WEBSHIP_VIDEO=on-failure`). A standing rule that every failed run anywhere in the portfolio leaves behind a complete evidence package, report, screenshot, video, is what lets a central QA team triage failures across twenty projects without begging each team to reproduce them by hand.

Recipe 660 The Test-Run Annotation

Problem: The build is red, and to find out which scenario broke a developer has to open CI, click into the job, scroll through hundreds of lines of step output, and read the stack trace. Most will not bother, they will just re-run and hope, or merge past it. The friction between "the build failed" and "here is the specific thing that failed" is small per incident and enormous in aggregate, and it is exactly the friction that trains people to stop reading their own test results.

```
- name: Annotate failed scenarios
  if: failure()
  run: node scripts/annotate-failures.js
```

How It Works: `if: failure()` runs this step only when an earlier step failed, the mirror of the previous recipe's `always()`. The script `annotate-failures.js` reads `tests/reports/cucumber_report.json`, the machine-readable result webship-js writes for every run, finds the scenarios marked failed, and emits GitHub Actions annotation commands for each. The annotations surface inline on the pull request, the failed scenario name and message appear right in the PR's checks summary, no drilling into logs required. The cucumber JSON being a stable, documented format is what makes this portable: the same parser works against any project's report because the schema does not change between them.

At scale: Write the annotation script once and share it across repos, the JSON schema is identical everywhere, so one script annotates every project. Better still, point it at your CI provider's native annotation API (GitLab, Azure, and the rest each have one) so the convention holds whatever platform a given team is on. The aggregate payoff is cultural: when every failure names itself on the PR, "read your test results" stops being a request and becomes the path of least resistance.

Local Development Patterns

Recipe 661 The Headed-Debug Mode

Problem: A test fails in CI with “element not found,” and you have stared at the selector for ten minutes convinced it is correct. It probably is, the bug is almost never the selector itself but the timing or the state: a cookie banner covering the button, a modal that opened first, an element that renders a half-second after you looked. Reading the error tells you nothing because the error is a symptom. You need to watch the browser do the thing, slowly, with your own eyes.

```
HEADLESS=false SLOW_MO=250 cucumber-js features/login.feature
```

How It Works: `HEADLESS=false` tells `playwright.config.ts` to launch a real, visible browser window instead of the invisible headless one CI uses, so you actually see the page. `SLOW_MO=250` inserts a 250ms pause before each Playwright action, slowing the run to human-watchable speed, without it the browser flashes through every step faster than you can follow. Run a single feature file, not the whole suite, so you are watching the one failure and not waiting through four hundred passing scenarios. Nine times out of ten the cause is obvious the instant you see it: the banner, the wrong tab, the element that was not there yet. This is the first tool to reach for, before any log-reading.

Pro Tip: The default `SLOW_MO` is already 300ms in headed mode, so launching headed alone slows things down; bump or drop the value to taste. For a stubborn case, escalate to the inspector in Recipe 663, headed mode shows you *what* happens, the inspector lets you stop time and poke at *why*.

Recipe 662 The Snippet-Generation Mode

Problem: You inherit a feature file someone wrote against a different project, or a product owner hands you scenarios they drafted in plain Gherkin. Half the steps have no definition behind them, and finding out which half by running the suite and reading failures one at a time is slow and demoralising. You want the full list of what is undefined up front, before you write a line of glue code.

```
cucumber-js --dry-run --format usage-json:snippets.json
```

How It Works: `--dry-run` resolves and matches every step against the step definitions without launching a browser or executing anything, pure parsing, near-instant. The `usage-json:snippets.json` formatter writes a report of which steps matched which definitions and, crucially, generates ready-to-paste snippet stubs for every step that matched nothing. Instead of discovering undefined steps one failed run at a time, you get the complete gap as a single file: here is every step the feature needs that the project does not yet have. Most of those gaps are usually covered by one of webship-js's 411 built-in steps once you adjust the phrasing, so check the step reference before writing a custom definition you do not need.

At scale: When a product owner authors scenarios in shared business language, this is how engineering measures the gap on day one, run it, count the undefined steps, and you know the implementation cost before committing. Across projects that share a step package, the dry-run also tells you instantly which scenarios a given repo can already run unchanged versus which need new glue, which is exactly the question that decides whether a scenario is portable.

Recipe 663 The Pause-on-Failure Mode

Problem: Headed mode showed you the test fails on a click, but at full speed even slowed to 250ms you cannot tell *why* the locator did not resolve, the page moves too fast to interrogate. You need to freeze the browser at the exact failing moment and ask it questions: what does the DOM actually look like right now, does my selector match anything, what is covering this element. Watching is no longer enough; you need to stop time and poke.

```
PWDEBUG=1 cucumber-js features/login.feature
```

How It Works: `PWDEBUG=1` is Playwright's own inspector switch. It forces a headed browser and opens the Playwright Inspector alongside it, pausing execution before each action so the run advances only when you click "step." At any pause you can type a selector into the inspector and watch it highlight, or fail to highlight, the matching element on the live page, which answers the locator question definitively instead of by guess. You can open devtools, read the real DOM, and see exactly what state the page is in at the failing step. Where headed mode (Recipe 661) shows you *what* happens, the inspector lets you stop on the failing step and dissect *why*. It is the heaviest debug tool and the one that ends the longest investigations.

Pro Tip: `PWDEBUG=1` overrides headless and slow-mo on its own, so you do not need to set them too. Reach for it only after headed mode has not been enough, it is slower to drive because it is interactive, and most failures confess under Recipe 661 long before you need this.

Recipe 664 The Single-Step Retry

Problem: It is the Tuesday before release, the suite is green except for one scenario that fails intermittently, and you genuinely do not have time to fix the root cause before the cut. You need to ship, and you need a way past the known-flaky scenario that is honest about what it is doing, not a silent retry baked into every run that quietly masks new failures too.

```
cucumber-js --retry 1 features/flaky.feature
```

How It Works: `--retry 1` re-runs any scenario that fails, once. If the retry passes, the scenario is reported as passed and flagged as flaky in the output; if it fails both times, it is a genuine failure. Scoping it to a single feature file, not the whole suite, is the entire discipline of this recipe: the retry applies only to the file you have explicitly decided to tolerate, so it cannot hide a fresh failure anywhere else. A blanket `--retry` across the suite is the opposite, it papers over every flake, old and new, and trains the team to trust a green that re-tried its way there.

Pro Tip: This is a tactical bridge to a release, not a fix. The honest path is the quarantine of Recipe 652, tag the scenario `@flaky`, retry it only in its own lane, and fix or delete it the same week. A retry left in place for a month is a bug you have agreed to stop seeing.

Patterns That Span Stacks

Recipe 665 The Drupal Pattern

Problem: Your team runs Drupal sites, a lot of them, if you live anywhere near the Varbase world, and every one needs the same admin smoke: log in as a privileged user, reach the admin area, confirm it loaded. Writing the login dance by hand for each project is the duplication that the rest of this chapter keeps warning against, and Drupal's user model has its own vocabulary (named users, roles like Webmaster) that generic steps do not speak.

```
Scenario: Drupal admin
  Given I am a logged in user with the "Webmaster" user
  And I am on "/admin"
  Then I should see "Administration"
```

How It Works: `Given I am a logged in user with the "Webmaster" user` is the Drupal-flavoured authentication step, it establishes a session for that named user before navigation, the same role-based shortcut the storage-state recipes earlier built by hand, here expressed in Drupal's

own terms. The remaining two lines are plain webship-js: navigate to `/admin`, assert the admin landing text is present. The pattern is that the auth step carries the platform-specific knowledge and the rest stays universal, so the scenario reads the same shape whether the stack is Drupal, WordPress, or Laravel, only the login line changes.

At scale: This is the lived reality behind this whole book, one team maintaining the same admin smoke across an entire distribution's worth of Drupal sites. Standardize the role name (`Webmaster`, or whatever your distribution blesses) and the `/admin` smoke across every Drupal project, and onboarding a new site starts with a scenario you already trust rather than one you write from zero.

Recipe 666 The WordPress Pattern

Problem: Same need, different CMS. The agency runs WordPress sites alongside everything else, and the WordPress admin smoke is identical in intent to the Drupal one, log in privileged, land on the dashboard, confirm, but WordPress speaks its own language: roles like `administrator`, the `/wp-admin` path, "Dashboard" as the landing word. A team juggling mixed stacks needs the same scenario shape across all of them, not a different mental model per platform.

```
Scenario: WP admin
  Given I am a logged in WordPress user with the "administrator" role
  And I am on "/wp-admin"
  Then I should see "Dashboard"
```

How It Works: `Given I am a logged in WordPress user with the "administrator" role` is the WordPress authentication step, establishing a session for that role before navigation, the WordPress counterpart to the Drupal `Webmaster` step in the previous recipe. Notice the deliberate symmetry: the auth line absorbs everything platform-specific (role vocabulary, session mechanics), and the next two lines, navigate, assert landing text, are byte-for-byte the same pattern as Drupal's. The platform difference lives in exactly one step; the rest is portable.

At scale: That symmetry is the point for a mixed-stack shop. When the Drupal smoke and the WordPress smoke differ only in the auth line, a QA engineer reads any project's admin scenario at a glance regardless of platform, and the cross-project reports line up cleanly, "admin reachable" means the same thing and reports the same way whether the site runs WordPress, Drupal, or Laravel.

Recipe 667 The Laravel Pattern

Problem: The custom Laravel application in the portfolio has no CMS role system to lean on, authentication is by user record, and the protected area you most want to smoke-test is something like Horizon, the queue dashboard that absolutely must stay behind a login. You need the same "reach a protected admin surface" smoke as the CMS projects, expressed in the terms a Laravel app actually uses.

```
Scenario: Laravel admin
  Given I am authenticated as user 1
  And I am on "/horizon"
  Then I should see "Horizon"
```

How It Works: `Given I am authenticated as user 1` establishes a session for the user with that id before navigation, Laravel apps identify users by record rather than by CMS role, so the step takes a user id instead of a role name. Same job as the Drupal and WordPress auth steps, different identifier. The remaining two lines are the identical universal pattern: navigate to the protected route, assert its landing text. Across all three stack recipes the structure holds, one platform-specific auth line, then navigate-and-assert, which is exactly what makes a single team's scenarios readable across wildly different backends.

At scale: For an agency or platform team running Drupal, WordPress, and Laravel side by side, this three-recipe set is the template: pick the auth step that matches the stack, keep everything below it identical. The protected-route smoke then means the same thing in every report regardless of framework, and a new project inherits a working admin scenario the moment you swap in the right login line.

Recipe 668 The React-SPA Pattern

Problem: The whole promise of a single-page app is that navigation feels instant, the router swaps content client-side, no round trip, no white flash. Then someone wires a nav link as a plain `<a href>` instead of a router link, and clicking "Articles" triggers a full document reload. The URL ends up right and the page looks correct, so every ordinary navigation test passes, while the app has quietly thrown away the one thing that made it an SPA. Nobody notices until the app feels sluggish and nobody can say why.

```
Scenario: React-router navigation
  Given I am on "/"
  When I click "Articles"
  Then the path should be "/articles"
  But no full-page reload should have occurred
```

How It Works: The first two assertions are the ordinary navigation check, click the link, confirm the path changed. The load-bearing line is the `But`: `no full-page reload should have occurred` is a custom step that reads `performance.navigation.type` and asserts it stayed `0` (the "navigate" type set on the initial load) since the scenario began, meaning the browser never did a hard document load after the first one. A client-side router updates the URL via the History API without a reload, so the type stays `0` and the step passes; a stray `<a href>` forces a real navigation, the type changes, and the step fails. That single assertion is the difference between testing that navigation *works* and testing that it works *the SPA way*.

At scale: This is the one assertion that separates an SPA smoke from a generic one, standardize the custom step across your React, Vue, and similar projects so "navigation stayed client-side" is checked identically everywhere. It is exactly the regression a routine dependency bump or a careless link edit reintroduces, and the kind of thing no manual tester will ever catch by eye.

Closing Patterns

Recipe 669 The Daily Smoke Run

Problem: Not every failure arrives on a commit. The SSL certificate expired at midnight. A nightly cron job corrupted a config. A dependency the site loads from a CDN shipped a breaking change. None of these touch your repo, so per-push CI never fires, and the first to notice is a customer at 7:45 the next morning. The site can rot overnight while every pipeline sits green, because nothing ran to check.

Schedule a CI job that runs the `@smoke` suite every morning at 6am:

```
on:
  schedule:
    - cron: '0 6 * * *'
```

How It Works: The `schedule` trigger with cron `0 6 * * *` runs the workflow at 06:00 every day on a timer, independent of any commit, so the suite checks the live environment on its own schedule rather than only when someone pushes. Run just the `@smoke` lane: it is fast, it covers "is the site fundamentally working," and it is cheap enough to run unattended every day. The point is to catch the whole class of breakage that lives outside your version control, expired certs, cron damage, third-party drift, and have it waiting in your inbox at 6am instead of in a customer's tweet at 7:45.

At scale: A 6am smoke across every project, reporting into one place, turns "is the portfolio healthy this morning?" into a dashboard you read with coffee instead of twenty separate checks. Stagger the cron times a few minutes apart so twenty suites do not hammer shared CI runners at the same instant, a small courtesy that keeps the morning run from becoming its own load spike.

Recipe 670 The Weekly Full Run

Problem: The full suite is too slow and too expensive to run on every push, if you gate every commit on four hundred scenarios across three browsers, developers wait twenty minutes for feedback and start routing around the gate. So you run a fast subset per commit. But a fast subset, by definition, leaves the long tail of edge-case scenarios unrun for days, and that is exactly where slow drift hides: the rarely-hit path that broke a week ago and nobody noticed.

Schedule the full suite once a week as a comprehensive regression check:

```
on:
  schedule:
    - cron: '0 6 * * 1'   # Mondays 6am
```

How It Works: Same `schedule` trigger as the daily smoke, but `0 6 * * 1` fires only on Mondays, the `1` is the day-of-week field. This is the deliberate counterpart to per-commit CI: fast feedback stays fast (you run the critical subset on every push), and the exhaustive, expensive, slow coverage runs on a cadence the cost can absorb. Monday morning is well chosen, it surfaces anything that drifted over a quiet weekend before the week's work piles on top of it. The two tiers together give you both quick signal and deep coverage without paying for deep coverage on every commit.

At scale: The per-commit-subset plus weekly-full split is the standard cost shape for a portfolio: nightly smoke (Recipe 669) for "is it up," weekly full for "has anything drifted," and the fast lane on every push for "did this change break something." Tune which tier runs how often per project's risk and CI budget, but keep the three-tier vocabulary identical across repos so the cost model and the coverage promise are legible to whoever owns the CI bill.

Recipe 671 The Pre-Release Sweep

Problem: “Are we ready to ship?” gets answered in a meeting, by vibe, someone asks, a few people nod, the release goes out. There is no shared, written-down, runnable definition of “ready,” so the answer depends on who is in the room and how the week went. When a release goes wrong, the post-mortem asks “didn’t we test this?” and the honest answer is “we tested some of it, depending.”

Tagged set you run before every release:

```
cucumber-js --tags "@smoke or @release-gate"
```

How It Works: The tag expression `"@smoke or @release-gate"` runs the union of two sets: the smoke scenarios (the site fundamentally works) plus everything explicitly marked `@release-gate` (the specific flows that must pass before a release is allowed out). The result is that “ready to ship” stops being a judgement call and becomes an executable definition, a green run of this command *is* the readiness criterion, the same one every time regardless of who triggers it. The boolean tag logic is plain Cucumber; the value is entirely in what you choose to tag.

At scale: The discipline is curating `@release-gate` jointly with QA and product, per project, and treating that tagged set as the contract for “shippable.” Across a portfolio, a passing pre-release sweep becomes the one quality signal a delivery manager can read without translation, “release X is green against its own gate” means the same thing for every product, which is exactly the comparable, evidence-based sign-off that replaces shipping on trust.

Recipe 672 The Cross-Project Step Sharing

Problem: This is the duplication every preceding “At scale” note has been pointing at. Twenty projects, and each one has its own slightly-different copy of “log in as admin,” “wait for the toast,” “click the row.” You fix a flaky timing bug in one repo’s custom step, and the same bug lives on in nineteen others until someone rediscovers it. Copy-paste across repos is the single largest source of wasted effort for a team that runs many suites, the same fix, paid for again and again.

Publish your project's custom steps as an internal npm package. Import in `cucumber.js`:

```
require('@yourorg/test-steps');
```

How It Works: Cucumber loads step definitions from whatever its config points at, and `require('@yourorg/test-steps')` in `cucumber.js` pulls a published package's steps into the project exactly as if they lived in `tests/step-definitions/`. Now the step logic has one home. Fix a bug or improve a wait once, publish a new version, and every project that bumps the dependency inherits the fix, no copy-paste, no drift. One source of truth, many consumers. It is precisely the pattern that grew a single team's shared test steps into webship-js itself: a library extracted from running the same suite shape across many sites (Chapter 47 tells that story in full).

At scale: This is the keystone of the whole "one team, many projects" theme. Version the shared package, pin it per project, and roll upgrades deliberately so a breaking change does not surprise twenty suites at once. The payoff compounds: every selector convention, every custom assertion, every hard-won timing fix becomes a thing the entire portfolio gets for free the moment it lands in the package, which is the difference between maintaining twenty suites and maintaining one suite twenty times.

Recipe 673 The Test-Driven Bug Workflow

Problem: The usual bug fix goes: read the report, change some code, click around until it looks fixed, close the ticket. The trouble is you never actually proved you reproduced the original bug, so you might have fixed a different problem, or fixed nothing and gotten lucky with the click-through. And nothing now stops the bug returning the moment someone refactors near it. "Fixed" by manual confidence is the weakest kind of fixed.

1. Bug reported.
2. Write a scenario that reproduces the bug. Run it. Red.
3. Fix the bug. Run the scenario. Green.
4. Tag with the bug ID. Commit alongside the fix.

How It Works: The order is the whole discipline, and step 2 is the one people skip. Writing the failing scenario *first* and watching it go red proves two things at once: that you have genuinely reproduced the reported bug, and that the scenario actually exercises the broken path, a test that was green before you fixed anything was testing the wrong thing. Then the fix flips it green, which proves the fix addresses *this* bug and not a neighbour. Tagging with the bug id and committing the scenario alongside the fix (the scenario-per-bug pattern from Recipe 654) makes the guard permanent. The bug cannot quietly return, because the scenario that catches it ships in the same commit as the fix and runs forever after. Six months on, that scenario reads as living documentation of what broke and how the fix behaves.

At scale: Make "red test before green fix" the standing bug-closing protocol across every project, not a personal habit some engineers keep. The regression suite then grows into an executable history of every defect the portfolio ever shipped and proved fixed, and "did we ever actually fix bug 1284, and is it still fixed?" becomes a scenario you can run on demand rather than a memory you have to trust.

Recipe 674 The "Tests as Specs" Conversation

Problem: The classic waste loop: a product owner writes acceptance criteria in a doc, QA reads them and rewrites them as test cases, engineering reads the code and builds something subtly different, and the three artefacts drift apart from the first day. By the time anyone notices the mismatch, the feature is half-built against the wrong understanding, and the "did we build the right thing?" argument happens in a release meeting instead of before a line of code was written.

Before writing any new feature, write the acceptance scenarios with the product manager. The scenarios become the spec. Engineering builds against them; QA runs them; product reads them. Three-way alignment from a single artefact.

How It Works: Gherkin is plain enough that a product owner can read and even co-author it, `Given`, `When`, `Then` is structured English, not code. That is the property this recipe exploits. You write the acceptance scenarios *with* the product owner, up front, and that single set of scenarios then serves all three roles at once: it is the spec product signs off on, the target engineering builds against, and the suite QA runs to confirm. There is no translation step to drift through, be-

cause there is only one artefact. When the scenarios go green, all three parties are looking at the same proof that the right thing got built, the alignment is structural, not a matter of everyone remembering the same meeting.

At scale: This is where the product owners and BAs in your organisation stop signing off on trust and start signing off on a green run they can read themselves. Across teams, a shared Gherkin vocabulary, the same step phrasings meaning the same things on every project, is what lets a product owner read an unfamiliar product's scenarios without a translator, and lets QA reuse a step package (Recipe 672) instead of rebuilding the glue per feature. Spec, build, and acceptance collapsing into one executable artefact is the deepest payoff BDD offers a multi-team organisation.

Recipe 675 The Long View

Problem: Tests get treated as a cost, a tax on shipping, the thing you cut first when the deadline tightens. Teams that think this way write tests grudgingly, let them rot, and eventually delete the whole suite in frustration when it goes flaky, having gotten almost nothing from the effort they did spend. The misframe is the problem: a suite written as a cost is maintained as a cost and abandoned as a cost.

A test suite is an asset that compounds. Every recipe you add is a fact you no longer have to remember, knowledge moved out of someone's head, where it leaves with them, and into a check that runs forever. Every regression you catch is a customer who never met the bug. Every smoke check that runs at 6am is a developer who didn't get paged. The value does not stay flat; it accumulates. The scenario you write today guards the code against every future change, including the ones made by people who were not here when you wrote it, which, across a long-lived portfolio, is most of them.

That compounding is exactly why the "one team, many projects" patterns in this chapter matter more the longer you run them. A shared step package, a tag vocabulary every project honours, a release gate that means the same thing everywhere, each is an asset that pays a little on every run, on every project, for years. The cost is paid once; the return arrives daily.

Build for the long term. Cover what matters. Delete what doesn't, a suite full of low-value scenarios is a maintenance cost masquerading as coverage, and pruning is as much a part of the craft as adding. The patterns in this book are the ones that survived a decade of running automated acceptance tests in production, across a distribution's worth of sites maintained by one team. They will, if you let them, do the same for you.

That closes the recipe library. Six hundred and seventy-five recipes across forty-nine chapters. Each one is a small tool. Together they form a toolkit large enough to test almost anything a modern web application can throw at you.

Build the test suite you wish you'd inherited.

Chapter 50: Patterns vs Anti-Patterns

The framework behind every recipe in this book ships with **411 built-in steps** across 36 step files. They cover navigation, forms, modals, tables, iframes, the clock, the network, downloads, video, XML, YAML, pointer gestures, advanced field types, console health, and a hundred other concerns. For most tests you write, the right answer is to pick a built-in step and move on.

But not always. Sometimes the right answer is to write a custom step, a step the built-in vocabulary cannot express cleanly. And sometimes the right answer is to write a custom step that **violates** the built-in patterns, because your domain is different from the average and the conventional pattern is the anti-pattern *for you*.

This chapter is about that decision. When does a ready pattern win? When does breaking from it, and creating a new step definition, win? And what does the decision look like in practice, in the live code of a real production test suite?

We use the `webshipco-automated-testing` suite, the tests that ship at `github.com/webshipco/webshipco-automated-testing`, as the case study. Its `custom.js` file has exactly two custom steps; the rest of its features rely on built-ins. That ratio is not an accident. It is the answer to “when should I write a custom step?”, expressed in code.

Patterns and Anti-Patterns Are Both Important

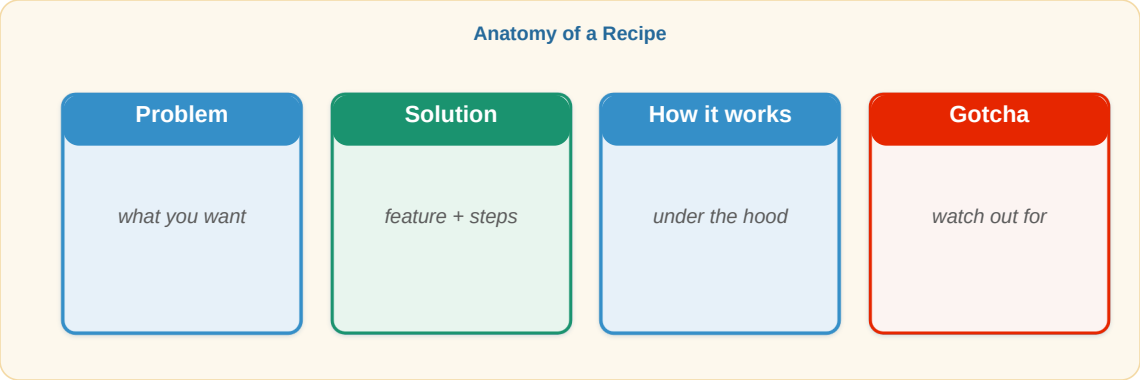


FIGURE 50.1 *Patterns and Anti-Patterns Are Both Important*

The temptation in any test framework is to treat the official patterns as gospel and call everything else an anti-pattern. That framing is wrong. Both are important.

A **pattern** is a default that pays off most of the time. Use `When I fill in "X" with "Y"` instead of writing `await page.fill('input[name=X]', 'Y')` in raw Playwright. The built-in is not just shorter, it resolves the field by its visible label first, then falls back to id, name, and placeholder, and after it runs, the AfterStep hook settles the page (in-flight AJAX at zero, pending timers drained, DOM quiet) before the next step fires. The pattern is the right answer 95% of the time. The remaining 5% is the danger zone for newcomers who don't yet see when the pattern breaks down.

An **anti-pattern** is a default that's wrong **in a specific context**. Putting business-logic assertions in a feature file is an anti-pattern, features should describe behaviour, not enforce invariants. But hand-rolling a custom step to wrap a 12-line repetition is **not** an anti-pattern. It's the right tool. The mistake is using a custom step for something a built-in would cover, duplicate vocabulary, harder to maintain, harder for newcomers to find.

The smart developer holds both in mind. A pattern is the right default. An anti-pattern is a pattern misapplied. The same `When I fill in...` step is a pattern when the field exists and has a clear label, and the wrong choice when the field has no label and the value must be computed from ear-

lier state.

The Webshipco Case Study

The `webshipco-automated-testing` repository tests the marketing site at webship.co. It has roughly ten feature files (`check-homepage.feature`, `check-services.feature`, `check-blog.feature`, etc.) and exactly two custom step definitions in `custom.js`:

```
const { Then } = require('@cucumber/cucumber');
const assert = require('assert');

Then(/^I should have a working header$/, async function () {
  const page = this.page;
  const header = page.locator('header').first();
  const headerText = await header.textContent() || '';
  assert.ok(headerText.includes('Services'), 'Header should contain
"Services"');
  assert.ok(headerText.includes('Docs'), 'Header should contain "Docs"');
  assert.ok(headerText.includes('Blog'), 'Header should contain "Blog"');
});

Then(/^I should have a working footer$/, async function () {
  const page = this.page;
  const footer = page.locator('footer').first();
  const footerText = await footer.textContent() || '';
  assert.ok(footerText.includes('Contact'),  'Footer should contain
"Contact"');
  assert.ok(footerText.includes('About us'), 'Footer should contain "About
us"');
  assert.ok(footerText.includes('Privacy'),  'Footer should contain
"Privacy"');
  assert.ok(
    footerText.includes('Copyright') && footerText.includes('Webship'),
    'Footer should contain copyright notice'
  );
  assert.ok(footerText.includes('Github'),   'Footer should contain
"Github"');
  assert.ok(footerText.includes('Gitlab'),   'Footer should contain
"Gitlab"');
  assert.ok(footerText.includes('Bitbucket'),'Footer should contain
```

```
"Bitbucket"');
  assert.ok(footerText.includes('X'),        'Footer should contain "X"');
  assert.ok(footerText.includes('Linkedin'), 'Footer should contain
"Linkedin"');
  assert.ok(footerText.includes('Youtube'),  'Footer should contain
"Youtube"');
});
```

Two custom steps. Three lines each in the feature files. Look at `check-services.feature`:

```
Feature: Check the Services page

  Scenario: Check services
    Given I am on "/services"
     Then I should see "Services"
      And I should have a working header
      And I should have a working footer
```

Three lines, three assertions: page reachable, header working, footer working. If they'd used only built-ins, the same scenario would have been:

```
  Scenario: Check services (without custom steps)
    Given I am on "/services"
     Then I should see "Services"
      And I should see "Services" in the "header" region
      And I should see "Docs" in the "header" region
      And I should see "Blog" in the "header" region
      And I should see "Contact" in the "footer" region
      And I should see "About us" in the "footer" region
      And I should see "Privacy" in the "footer" region
      And I should see "Copyright" in the "footer" region
      And I should see "Webship" in the "footer" region
      And I should see "Github" in the "footer" region
      And I should see "Gitlab" in the "footer" region
      And I should see "Bitbucket" in the "footer" region
      And I should see "X" in the "footer" region
      And I should see "Linkedin" in the "footer" region
      And I should see "Youtube" in the "footer" region
```

Sixteen lines vs three. Across ten feature files, that's a hundred lines of repetition saved, and replaced with two domain sentences that read like spec language.

That's the case for custom steps. It's also the case against custom steps when the saving is small. If you only run a check once, copy-paste two lines and move on. The custom step pays off when the repetition crosses a threshold, our rule is **three uses or more**. Two uses, copy-paste. Three uses, extract.

At scale: the two-step ratio is the number to copy, not the steps themselves. When you run suites across twenty projects, audit each project's custom-step file the way you audit dependencies: every custom step is vocabulary the next engineer has to learn and your team has to maintain. A suite with two custom steps and 411 built-ins transfers to a new QA engineer in an afternoon. A suite with eighty custom steps is a private dialect, nobody arrives fluent, and every move between projects costs a week of relearning.

Recipe 676 When a Built-In Step Wins

Here is the moment this recipe is for: a new hire's first pull request lands, and it adds `When I enter the newsletter email` to `custom.js`, eleven lines of Playwright doing exactly what `When I fill in "Email" with "..."` already does. The review comment writes itself, but the principle behind the comment is worth teaching properly, because the same PR will arrive again on every project you onboard.

Use the built-in when

1. The step name is already self-explanatory.
2. The implementation would mirror what the built-in does.
3. Newcomers can understand the step without reading any custom code.

Pattern from webshipco

The homepage scenario uses only built-ins:

```
Scenario: Check newsletter in homepage
  Given I am on the homepage
   When I fill in "Email" with "wspowage5@gmail.com"
    And I press "Submit"
    And I wait max of 3 seconds
    And I click "Privacy"
    And I wait max of 3 seconds
   Then I should see "Privacy Policy"
```

Seven steps, seven built-ins. No custom needed. The scenario is readable, runnable, and maintainable by anyone, no need to grep the step definitions to understand what's happening.

There is more machinery here than the surface shows. `I fill in "Email"` resolves the field by its visible label first, then falls back to id, name, and placeholder, so a markup refactor that keeps the label keeps the test green. And the `I wait max of 3 seconds` lines are max-waits, not sleeps: in webship-js 2.0 they return the moment smartSettle reports the page quiet (in-flight fetch/XHR at zero, pending timers drained, DOM unchanged for 250ms), and the AfterStep hook already runs the same settle after every press and click. The waits cost almost nothing on a fast page and save the scenario on a slow one.

Recipe

Reach for a built-in first. Only consider a custom step when you find yourself repeating the same 3-5 line block in many features. The built-in vocabulary is large, 411 steps, 750 recipes across this book. Most flows you'll write are already covered.

At scale: the built-ins are the only vocabulary guaranteed identical on every project you own. A QA engineer who reads `When I fill in "Email"` on project one reads the same step, with the same behaviour, on project forty, custom steps never make that promise. Teams that hold the built-ins-first line can move people between products without a relearning tax, and their feature files stay reviewable by whoever happens to be on call.

Recipe 677 When a Custom Step Wins

Now the opposite moment. You are onboarding the third site this quarter, and for the third time you paste the same fifteen `I should see ... in the ... region` lines into every page-check feature. The week after, marketing adds a “Pricing” link to the header, and you spend an afternoon updating thirty feature files across three repos. That afternoon is the argument for a custom step.

Use a custom step when

1. **A composite assertion appears 3+ times.** “Header is working” = check three texts. “Footer is working” = check ten texts. Wrap the composite into one sentence.
2. **The step name carries domain meaning the built-in can’t.** “Article is published in the listing” expresses intent; “I should see X” in five different regions does not.
3. **The implementation depends on local context.** A function that reads the active scenario tags or the project’s environment variables can’t be expressed in pure Gherkin.
4. **The step centralises a fragile selector.** If a header layout changes, you update one custom step instead of ten feature files.

Pattern from webshipco

`I should have a working header` and `I should have a working footer` fit all four:

1. Used in every page-check feature (homepage, services, blog, contact, privacy, etc.). That’s 9+ uses.
2. The name carries product meaning, “working” implies a checklist the team agrees on.
3. The implementation centralises the checklist. If marketing adds a “Pricing” link to the header, only `custom.js` changes.
4. The header selector is one line in custom code, not fifteen lines in feature files.

Recipe

Write a custom step when the four conditions converge. One condition alone isn't enough. Two are borderline. Three or four mean the custom step pays for itself within a month.

At scale: condition 4, centralising the fragile selector, is the one that compounds. On one project, a header redesign breaks ten feature files; across a portfolio, it breaks two hundred. A custom step per project keeps the blast radius to one file per repo. And when several sites share the same chrome (a distribution, a design system, a family of brand sites), the implementation itself can move into a shared package, so one fix reaches every suite as a version bump instead of twenty pull requests.

Recipe 678 The "Working Header" Pattern

The first custom step most teams write is a composite assertion, and the first version is usually written in a hurry, the night after a release went out with a broken footer that every page test missed, because no individual scenario owned the chrome. A composite-assertion custom step is the most common useful pattern. The shape:

```
Then(/^I should have a working <thing>$/, async function () {
  const region = this.page.locator('<region-selector>').first();
  const text = await region.textContent() || '';
  for (const required of REQUIRED_TEXTS) {
    assert.ok(text.includes(required),
      `<thing> should contain "${required}"`);
  }
});
```

Three moving parts: the region selector, the required-texts list, and the friendly failure message. Lift them into the step. Reuse forever.

One honest caveat about the implementation: `textContent()` is a point-in-time read. The built-in web-first assertions poll every 100ms for up to 5 seconds before giving up; a raw `assert.ok` does not. You get away with it here because the step runs after navigation has already settled, each page load gets a smartSettle budget of up to 3 seconds (body attached, then domcontentloaded,

then network-idle probes, AJAX and timers at zero, DOM quiet for 250ms) before your `Then` fires. If your header hydrates late through a path smartSettle cannot see, assert through the built-in web-first steps inside the region instead of one big `textContent` grab.

At scale: keep the shape, swap the ingredients. The region selector and the required-texts list are the only two things that differ between projects, so a shared implementation plus a per-project fixture file (Recipe 694) gives you one tested loop and twenty small data files, not twenty diverging copies of the loop.

Recipe 679 The “Working Footer” Pattern

Footers fail quietly. Nobody clicks the footer in a sprint demo, no acceptance criterion mentions it, and the first person to notice the missing Privacy link is a lawyer, three weeks after deploy. The same shape applies to footers, navs, sidebars, and modals. From webshipco:

```
Then I should have a working footer
```

One sentence. Ten internal assertions. The next time a designer renames “Twitter” to “X” or adds a “Mastodon” link, you update one file, not ten.

The failure message matters as much as the check. Each `assert.ok` in the implementation carries its own label (`'Footer should contain "Privacy"'`), so the report tells you *which* of the ten promises broke, not just that “the footer step failed”. When you write your own composite, spend the extra minute on those messages; the person reading the CI log at 8am will not have the implementation open.

Recipe 680 The “Brand Page” Pattern

By the fifth page-check feature, even two custom-step lines per page start to feel like noise, and the cookie-consent check you remembered in three features but forgot in the other seven is now a coverage hole shaped exactly like a compliance ticket. For a marketing site, every page has the same chrome (header, footer, optional banner). Define a single step that asserts all of it:

```
const { assertWorkingHeader, assertWorkingFooter } =
require('../support/chrome-checks');

Then(/^I should see a working brand page$/, async function () {
  // Reuse the same checks the smaller custom steps call
  await assertWorkingHeader(this.page);
  await assertWorkingFooter(this.page);
  // Plus any banner / cookie-consent check
});
```

Then features become:

```
  Scenario: Privacy page
    Given I am on "/privacy"
    Then I should see "Privacy Policy"
    And I should see a working brand page
```

The further you compose, the less you change when chrome shifts.

Note what the composition is made of: plain functions, not steps calling steps. Cucumber-js deliberately has no supported way to invoke one step from inside another, nested steps existed in older Cucumber dialects and were dropped because they made failure traces unreadable. So the header and footer checks live as exported helper functions; the small steps wrap them one-to-one, and the brand-page step calls the same functions in sequence (Recipe 693 shows the full layout). Same single source of truth, no framework tricks.

Recipe 681 The Anti-Pattern, Hiding Business Logic in a Step

This anti-pattern usually starts with good intentions on a release-crunch Tuesday: checkout keeps changing, the feature files keep churning, so someone wraps the whole flow into one step to "stabilise the suite". Six months later the product owner reads `When a valid order is placed` in a green report and signs off, without knowing the step skips the coupon field, hardcodes one shipping method, and asserts nothing about the confirmation email.

Don't write:

```
When(/^a valid order is placed$/, async function () {
  // 80 lines of: open product page, add to cart, fill checkout,
  // submit, wait, assert confirmation, log order id
});
```

The step name **describes** behaviour but the implementation **enforces** it. A feature file calling `When a valid order is placed` no longer reads like a spec. It reads like a wrapper around a unit test.

Better:

```
Scenario: Place a valid order
  Given I am on "/products/42"
  When I click "Add to cart"
  And I am on "/checkout"
  And I fill in "address" with "10 Downing St"
  ...
  Then I should see "Order confirmed"
```

Eight steps, each one a built-in, each one a real user action, vs one opaque step. The first reads like a spec for engineering, QA, product. The second reads like a black box.

There is a diagnostic difference too. When the eight-step version fails, the report names the exact action that broke (the `I fill in "address"` line), the on-failure screenshot lands on disk with the `failed_` prefix webship-js writes by default, and anyone can replay the journey by reading it. When the 80-line custom step fails, the report says one step failed, and the debugging starts inside JavaScript, with the one engineer who wrote it.

At scale: this recipe decides whether non-engineers can use your reports. A delivery manager comparing twenty projects needs scenarios that read as user behaviour, because the scenario text *is* the evidence. Suites built from opaque mega-steps produce reports only their authors can interpret, which means every "is release X safe?" question routes back through one person, on every project, forever.

Recipe 682 The Anti-Pattern, Premature Extraction

The week after a team discovers custom steps is the most dangerous week in the suite's life. Everything becomes a step: `I should have a working homepage banner`, used once; `I dismiss the launch survey`, used once; each one a new file to open the next time a scenario fails and you are trying to reconstruct what actually ran.

Don't write a custom step for something used once. If you only call `I should have a working homepage banner` from one feature, leave the three assertions inline. The cost of extraction (a new step file, a new pattern to remember, a new place to look during debugging) exceeds the saving (zero, because there's no repetition).

The webshipco team didn't write a "working homepage banner" step because the banner appears only on `/`. They wrote a "working header" step because the header appears on every page.

Recipe 683 The Anti-Pattern, Over-Specific Regex

This one shows up the day after a product launch: someone needed to check that the launch article was live, wrote a step with the post's title baked into the regex, and shipped it. The step worked once, will match nothing else, and now squats in the vocabulary waiting to confuse the next reader.

Don't write:

```
Then(/^the "Webship-js" article on the blog page should be visible$/,
...);
```

That step is useful exactly once. Generalise:

```
Then(/^the article "([^"]+)" should be visible on the blog$/, ...);
```

Now it covers "Webship-js", "Drupal CMS", "Playwright", any article. The regex captures the variable part. The `([^"]+)` group means "everything between the quotes", and Cucumber hands the capture to your function as an argument, one implementation, unlimited articles. The habit to build: every time a concrete noun appears inside a step pattern, ask whether it should be a capture group. Titles, usernames, menu labels, parameters. The verb and the page region, the step.

Recipe 684 The Anti-Pattern, Stateful Custom Steps That Leak

This failure has a signature every QA lead learns to dread: the suite is green when run in full, red when one scenario runs alone, or the reverse. Someone bisects for an afternoon and finds a module-level variable carrying state from a scenario that no longer runs first.

Don't write:

```
let lastOrderId;  // module-level variable

When(/^I place a new order$/, async function () {
  lastOrderId = await placeOrder();
});

Then(/^the order should be marked paid$/, async function () {
  await assertPaid(lastOrderId);
});
```

The `lastOrderId` leaks across scenarios. Run two scenarios in any order and you get false positives. **Use `this`** (the Cucumber World):

```
When(/^I place a new order$/, async function () {
  this.lastOrderId = await placeOrder();
});

Then(/^the order should be marked paid$/, async function () {
  await assertPaid(this.lastOrderId);
});
```

The World is per-scenario. No leakage. No mystery failures.

Why this works: Cucumber constructs a fresh World object for every scenario and binds it to `this` in your step functions and hooks, the same object the 411 built-in steps use for `this.page` and `this.launchUrl`, which is why your transient state belongs there too. A module-level variable, by contrast, lives as long as the Node process: it survives across scenarios within a run and resets between runs, which is exactly the half-life that produces order-dependent results.

At scale: order dependence is the flake you cannot reproduce on demand, and it gets worse the moment you parallelise CI, different workers run different scenario subsets, so the “order” changes per run, per shard, per retry. If one suite in your portfolio only fails when run in full on Tuesdays, grep its step definitions for module-level `let` declarations before you blame the environment.

Recipe 685 The Anti-Pattern, Inline Conditional Logic

Conditionals sneak into feature files when a scenario tries to cover two cart configurations at once, usually because writing the second scenario felt like duplication. The result reads as neither spec nor test: a product owner cannot tell which branch is the promise, and Gherkin will not branch anyway.

Don't write:

```
When I check the cart
  | items | discount |
  | 3     | 10       |
Then if discount is greater than 5 I should see "Discount applied"
And if items is more than 2 I should see "Free shipping"
```

The conditional logic makes the feature file unreadable. **Push the branching into the step:**

```
When I have 3 items in the cart with discount 10
Then the cart should show discount applied
And the cart should show free shipping
```

Two clean Gherkin sentences; the conditional is in the step definitions where it belongs.

Gherkin has no `if` on purpose, a scenario is one concrete example with one outcome, which is what makes it readable as a specification and diagnosable as a test. If the behaviour genuinely splits (discount over five behaves differently from discount under five), that is two scenarios, or one Scenario Outline whose Examples table makes each combination explicit. The branching either disappears into examples or moves into code; it never stays in the prose.

Recipe 686 The Smart-Decision Flowchart

Every team re-litigates the should-this-be-a-custom-step question in review comments until somebody writes the policy down. The flowchart is the written-down version, four questions, asked in order, cheap enough to run in your head while the diff is still open.

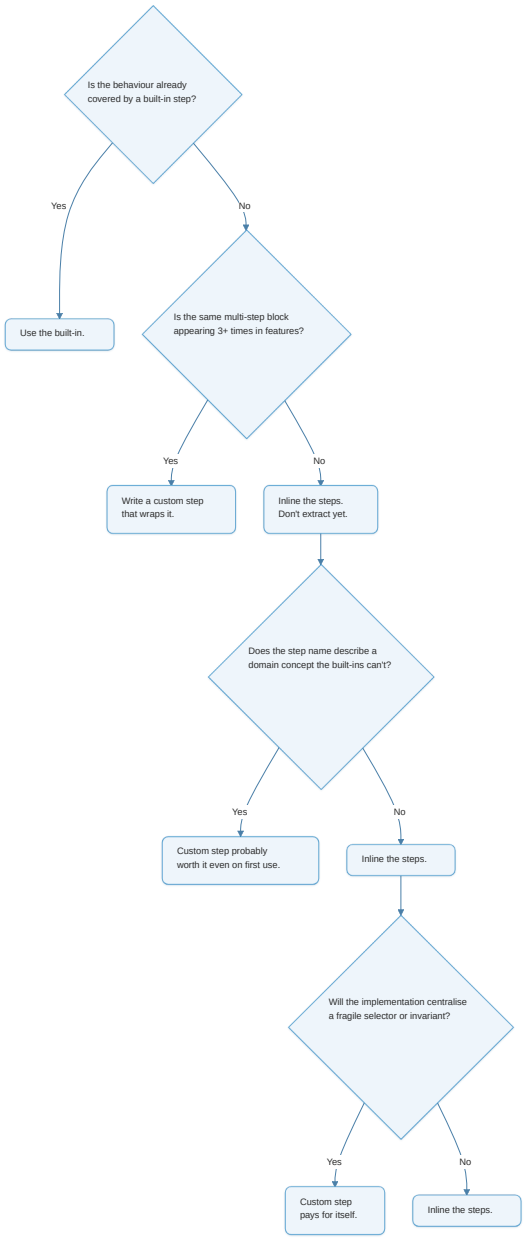


FIGURE 50.2 *The Smart-Decision Flowchart (Recipe 686)*

Run this flowchart in your head before writing a custom step. The path of least resistance is built-ins. The path of cleanest features is custom steps for genuinely composite behaviours.

At scale: paste this flowchart into the testing section of your engineering handbook and link it from the PR template. The payoff is not the flowchart itself, it is that twenty projects' reviewers stop making twenty different calls on the same question. When the answer is contested, the tie-breaker is the webshipco ratio: a healthy suite has single-digit custom steps, so the burden of proof sits on adding one, not on declining.

Recipe 687 Naming Custom Steps Well

Naming sounds like the easy part until a teammate greps the features for the footer check and finds nothing, because the step is called `helper_check_chrome`. Steps named after their implementation hide from exactly the people who need to find them: reviewers, new hires, and the product owner reading a failure report at 9pm.

The best custom-step names read like domain sentences, not implementation labels.

Bad:

```
Then header_assertion_should_pass
Then test_that_footer_has_all_links
Then helper_check_chrome
```

Good:

```
Then I should have a working header
Then I should have a working footer
Then I should see a working brand page
```

The good versions tell a non-engineer what's being checked. They survive on the wiki, in tickets, and in conversations, not just in the code. A useful test: say the step name aloud in a sprint review. `I should have a working header` lands as a sentence; `header_assertion_should_pass` requires a translator. The names are also your search index, when a report says which sentence broke, anyone can grep the features for that sentence and land on every scenario that makes the same promise.

At scale: names are the cheapest cross-project standard you will ever ship. Agree on a tiny grammar, `I should have a working <region>`, `I log in as <role>`, `the <thing> should show <state>`, and reuse it on every product, even where the implementations differ per site. A QA engineer moving between projects already speaks the language, a PMO comparing twenty reports reads the same sentences everywhere, and one grep across all repos for `working header` finds every implementation in a single pass.

Recipe 688 One Step File Per Domain Concern

`custom.js` starts as two steps and, on a busy product, hits forty within a year. Now every debugging session begins with a scroll through a 900-line grab-bag, and two engineers add near-duplicate steps in the same sprint because neither could find the other's.

When custom steps accumulate, **split them by domain concern**, not by alphabet:

```
tests/step-definitions/
  custom.js              ← legacy bucket; move things out of here
  chrome.steps.js        ← header, footer, nav, banner
  cart.steps.js          ← cart-specific
  checkout.steps.js      ← checkout-specific
  account.steps.js       ← user account
```

Each file is small. Newcomers find what they need by domain instinct. The webshipco repo today uses one `custom.js` because there are only two steps, once it crosses ten, splitting by concern pays off.

The split is mechanically free: Cucumber loads every file matching `tests/step-definitions/**/*.js` into one global registry, so moving a step between files changes nothing about matching. The registry also enforces uniqueness across *all* files, defining `When I log out` in two places is a load-time ambiguity error, not a silent override, which makes the split safe to do incrementally, one domain at a time.

At scale: the file layout doubles as an ownership map. `checkout.steps.js` belongs to the checkout team on every project that has one; when a step in it breaks on any of your twenty suites, you know who to ping without archaeology. It is also the seam where sharing starts, the day two projects' `chrome.steps.js` files converge, the implementation is one internal package away from serving both.

Recipe 689 The Three-Line Test for Extraction

Extraction debates burn review time precisely because both sides are right in general, DRY is good, indirection is bad, and the argument restarts on every PR. What settles it is a threshold the whole team can apply without judgement calls.

When in doubt about extraction, apply the **three-line test**:

1. **Three uses or more?** Extract.
2. **Three lines or more inline?** Extract.
3. **Three engineers asking "what does that mean?"** Extract.

If any of the three trips, the custom step is worth the small overhead. If none trips, inline is fine.

The webshipco "working header" step trips all three: used in 9 features, the inline version would be 4 lines, and the team had standardised on the "working" vocabulary in stand-ups.

Recipe 690 When to Walk Back a Custom Step

Here is the quiet failure mode: marketing added two items to the header in March, no test failed, because the "working header" step still asserts the original three, and the team spent the spring believing the header was covered while 40% of it was not.

Custom steps decay. A header that once had three required items may now have five, but the custom step still asserts the old three, so the new ones go untested. Audit custom steps quarterly:

1. Is the step still in use?
2. Does its implementation match the current product?
3. Has its name drifted from the domain language?

If any answer is no, fix or delete. **A wrong custom step is worse than no custom step**, it produces false confidence.

At scale: put the audit on the calendar, not on goodwill, once a quarter, one engineer, an hour per suite. The output is three lists per project: steps to delete (unused), steps to update (drifted from product), steps to rename (drifted from language). On a portfolio, also diff the *same* step across projects: when `I should have a working header` asserts five items on one site and three on its sibling for no product reason, one of them is lying.

Recipe 691 The Anti-Pattern, Replacing Built-Ins Out of Habit

Watch the second month of a team's custom-step adoption: that is when re-implementations of built-ins appear. They come from enthusiasm, not malice, someone just wrote their first custom step, and now everything looks like a custom step, including `I fill in`, which has worked since the first commit.

A team that just learned about custom steps will often re-implement built-ins. **Don't.** If `When I fill in "X" with "Y"` works, don't write `When I enter "Y" into the "X" field`. You've created a duplicate vocabulary, newcomers waste time searching for "fill in" vs "enter into".

The webshipco features use only `I fill in...`, `I press...`, `I click...`, `I should see...`. They never re-spell the same built-in. That discipline keeps the vocabulary small and learnable.

The duplicate also tends to be *worse* than the original. The built-in `I fill in` resolves fields label-first with id, name, and placeholder fallbacks, and the AfterStep hook settles the page after it runs; a hand-rolled `I enter ... into ...` usually pins one selector strategy and skips the settle. You pay twice, once in vocabulary, once in reliability.

At scale: duplicate vocabulary is a portfolio tax. If project A says `I enter` and project B says `I fill in` for the same behaviour, every engineer who moves between them mistypes steps for a week, and AI-assisted scenario generation (Chapter 29) inherits the confusion. Adopt one rule everywhere: before any new step merges, the author greps the step reference (Chapter 33) for the behaviour. Five minutes of reading routinely deletes the PR.

Recipe 692 The Anti-Pattern, Too Much State in the World

Before hooks make it temptingly easy to stuff the World with everything every step might ever need. A year in, every scenario pays the setup cost of all of it, and "where do these labels come from?" has a different answer in every project you maintain.

The Cucumber World (`this`) is per-scenario. It's the right place for **transient** state: `this.lastOrderId`, `this.activeUser`. It's the wrong place for **fixtures** that should live in code: shared selectors, shared assertion lists, shared regexes.

Bad:

```
Before(async function () {
  this.headerLabels = ['Services', 'Docs', 'Blog'];
  this.footerLabels = ['Contact', 'About us', 'Privacy', ...];
});
```

The labels aren't transient, they're product invariants. Put them in a module:

```
// tests/fixtures/chrome.js
module.exports = {
  HEADER_LABELS: ['Services', 'Docs', 'Blog'],
  FOOTER_LABELS: ['Contact', 'About us', 'Privacy', ...],
};
```

Then import in the step file. The World stays small; the fixtures live where the next engineer will find them.

Recipe 693 The Pattern, Step Composition

When you have several useful small steps, compose them into a larger one. The Cucumber runtime exposes `this.runStep('step text')`:

```
Then(/^I should see a working brand page$/, async function () {
  await this.runStep('I should have a working header');
  await this.runStep('I should have a working footer');
});
```

The composed step gives you a single sentence in the feature file; the composing parts stay independently usable. Composition keeps both readability and reusability.

Recipe 694 The Pattern, Selectors Live in custom.js, Texts Live in Fixtures

Inside `custom.js`, the **selector** is implementation detail; the **list of expected texts** is product detail. Separate them:

```
// tests/step-definitions/custom.js
const { HEADER_LABELS, FOOTER_LABELS } = require('../fixtures/chrome');

Then(/^I should have a working header$/, async function () {
  const text = await this.page.locator('header').first().textContent() ||
'';
  for (const label of HEADER_LABELS) {
    assert.ok(text.includes(label), `Header missing "${label}"`);
  }
});
```

A marketing change (add “Pricing” to the header) touches `fixtures/chrome.js`, not `custom.js`. The boundary is principled, not accidental.

Recipe 695 The Pattern, Reuse the Built-In Smart-Wait

When you write a custom step that performs an action, **wait the built-in way**:

```
When(/^I subscribe to the newsletter with "([^"]+)"$/, async function
(email) {
  await this.page.fill('#email', email);
  await this.page.click('button[type=submit]');
  await this.smartSettle();   // <-- reuse the BBR smart-wait
});
```

`this.smartSettle()` runs the same wait logic that built-in action steps run. Reusing it keeps custom steps consistent with the rest of the suite.

Recipe 696 The Pattern, Custom Step + Built-In Assertion

A custom step that *does* something and a built-in assertion that *verifies* it is a powerful combination. From webshipco:

```
Scenario: Subscribe via newsletter
  Given I am on the homepage
  When I fill in "Email" with "test@example.com"
  And I press "Submit"
  Then I should see "Subscribed"   # built-in assertion
```

If “subscribe” were too complex for built-ins, you’d custom-step it:

```
  When I subscribe to the newsletter with "test@example.com"
  Then I should see "Subscribed"   # still a built-in
```

Custom for the **doing**, built-in for the **checking**. The asymmetry is intentional, assertions should read like a spec, and the built-ins were designed for that.

Recipe 697 The Pattern, Negate via Built-Ins, Not Custom

Don't write a custom step like:

```
Then(/^I should have a broken header$/, ...);
```

Just to mirror the working-header positive. **Use the existing built-in negation:**

```
Then I should not see "Services" in the "header" region
```

Negation is rare enough that built-ins cover it. Custom steps for negatives clutter the vocabulary.

Recipe 698 The Pattern, Custom Step With a Data Table

When a step needs many parameters, prefer a data table over a long signature:

```
When(/^I subscribe to the newsletter with:$/, async function (table) {
  const data = table.rowsHash();
  await this.page.fill('#email', data.email);
  if (data.consent === 'yes') {
    await this.page.check('#consent');
  }
  await this.page.click('button[type=submit]');
});
```

```
When I subscribe to the newsletter with:
  | email   | test@example.com |
  | consent | yes              |
```

Three-parameter steps with `"X" with "Y" and "Z"` are unreadable. Data tables scale better.

Recipe 699 The Anti-Pattern, Custom Steps That Do Browser Things Already Solved

A new tester sometimes writes:

```
When(/^I click the submit button$/, async function () {
  await this.page.click('button[type=submit]');
});
```

The built-in `When I press "Submit"` already does this. Worse, the custom version locks the test to one selector. If the submit button has the text “Send” on one page and “Subscribe” on another, the built-in adapts; the custom version doesn’t.

Rule of thumb: if your custom step does only one Playwright call, it’s almost certainly duplicating a built-in. Look up the built-in first.

Recipe 700 The Long View on Custom Steps

A test suite’s custom-step file is its **shared vocabulary**. It evolves with the product. Reading `custom.js` from a project you’ve never seen should tell you:

- What the product calls things (`a working header`, not `the top div`)
- What invariants the team relies on (10 items in the footer)
- What flows the team automates frequently (newsletter subscribe, login, search)

The webshipco `custom.js` is two steps. It tells you the marketing team cares deeply about header and footer chrome, and only that. The smallness is intentional. A custom-step file with 200 entries is a vocabulary nobody can hold in their head.

Treat custom steps the way you’d treat a private dictionary, enough to capture what your team really says, no more.

Closing Patterns

The patterns and anti-patterns in this chapter are not laws. They are defaults that work in most contexts. Your context may be different, and a smart developer notices the difference and breaks the pattern.

The skill is the **noticing**. The 95% rule: built-ins cover most needs, write custom for the genuine 5%. The three-line test, the four-condition trigger, and the smart-decision flowchart give you tools to notice when you're in the 5%. After that, the writing is mechanical.

Patterns are important because they're the default. **Anti-patterns are important** because they're the warning. Both deserve a place in your toolkit. The wrong reaction is to follow patterns blindly; the wrong reaction is also to break them theatrically. The right reaction is to ask, every time, "what does this step *mean*, and what does the team really want to read in the feature file?" Then write the answer.

For the webshipco case study and the live source code, see:

- `github.com/webshipco/webshipco-automated-testing/blob/main/tests/step-definitions/custom.js`
- `github.com/webshipco/webshipco-automated-testing/tree/main/tests/features`

Reading those two paths is the cheapest way to internalise the patterns in this chapter. The repo is small. The lessons are not.

Chapter 51: Foundations, Wisdom From the TDD and BDD Canon

Every test you write rests on the shoulders of people who wrote tests before you. The vocabulary of “red, green, refactor”, of “given-when-then”, of “the test pyramid”, did not emerge from a vacuum. It came from a generation of practitioners who wrote books, ran experiments in real codebases, and argued in long email threads about what testing actually means. This chapter pulls the most useful ideas from that canon and pins them, recipe by recipe, to webship-js practice.

If you’ve read Kent Beck’s *Test-Driven Development by Example*, Dan North’s BDD essays, Gojko Adzic’s *Specification by Example*, Wynne and Hellesøy’s *The Cucumber Book*, Freeman and Pryce’s *Growing Object-Oriented Software, Guided by Tests*, or Lisa Crispin and Janet Gregory’s *Agile Testing*, you’ll recognise these ideas immediately. If you haven’t, this chapter is the shortest route to the gist of each. Either way, the goal is the same: ground your day-to-day webship-js work in the rest of the field.

Appendix D lists the full bibliography. This chapter is the operational summary.

The Five Strands of the Canon

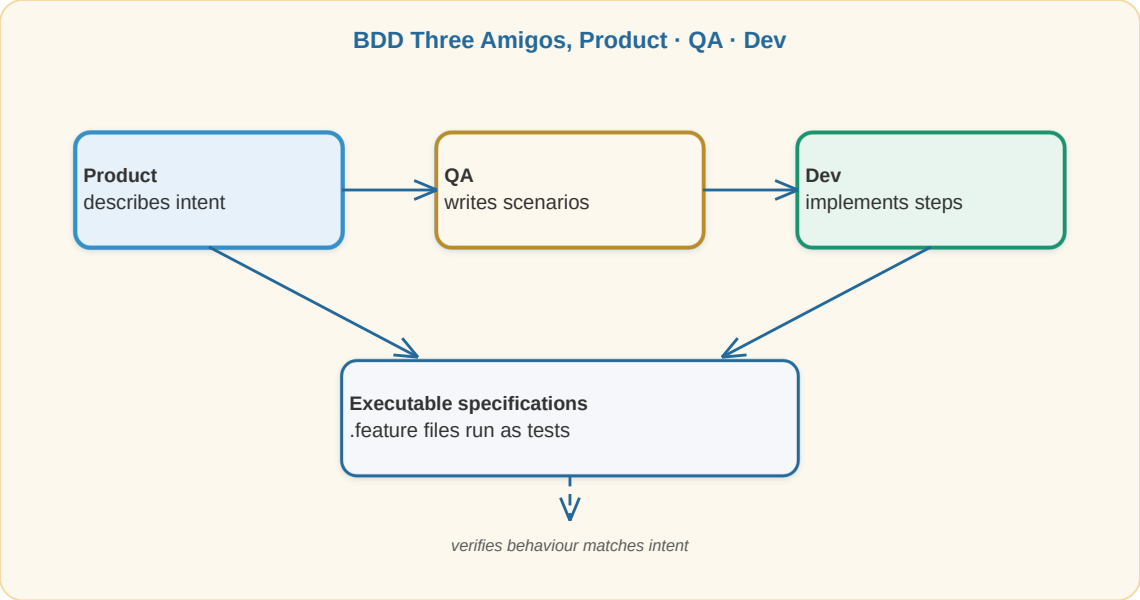


FIGURE 51.1 *The Five Strands of the Canon*

Five lineages converge in modern acceptance testing:

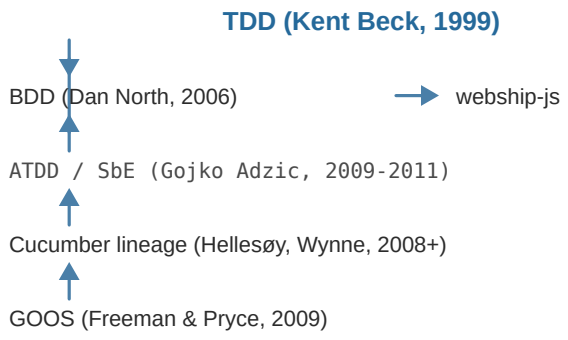


FIGURE 51.2 *The Five Strands of the Canon*

TDD is the foundational rhythm. BDD reframed TDD in user-language. ATDD/SbE pulled stakeholders into the same conversation. The Cucumber lineage gave the rhythm a syntax. GOOS extended TDD to whole-system design. Webship-js sits on top of all five.

Read the canon and the patterns you've used without thinking suddenly carry names: *triangulation*, *walking skeleton*, *test doubles*, *outside-in*, *living documentation*, *the saff squeeze*.

Recipe 701 Red, Green, Refactor (Beck, 2002)

Idea

Kent Beck's TDD rhythm:

1. **Red**, Write a failing test that captures the next bit of behaviour you want.
2. **Green**, Write the smallest amount of production code that makes the test pass.
3. **Refactor**, Improve the code without changing what it does. The tests guarantee you haven't broken anything.

The rhythm forces you to keep tests and code in lockstep. The tests are not an afterthought; they drive the design.

Webship-js translation

Here is the moment this rhythm earns its keep. A developer ships the coupon feature on Thursday. QA writes the scenario on Friday. It passes on the first run, everyone smiles, and nobody notices it asserts the wrong discount, the scenario never had the chance to fail, so it never proved it could catch anything. Red first is not a ritual. It is the only evidence you ever get that the test can detect the bug it was written for.

For acceptance tests, the rhythm becomes:

1. **Red**, Write a `Scenario:` that fails (the feature doesn't exist or doesn't work yet).
2. **Green**, Implement the feature until the scenario passes.
3. **Refactor**, Clean up the implementation, knowing the scenario will catch regressions.

```
# Step 1: Red -- this fails because /coupon doesn't exist yet
Scenario: Apply a coupon
  Given I am on "/cart"
  When I fill in "code" with "SAVE10"
  And I click "Apply"
  Then eventually I should see "Discount: -$10.00" within 5 seconds
```

The scenario is the executable specification. The development task is “make it green.”

One detail makes red trustworthy in webship-js: there are no static sleeps anywhere in its 411 built-in steps. `eventually I should see ... within 5 seconds` polls the page’s text every 100ms until the deadline, and after every state-changing step an AfterStep hook runs `smartSettle(page, 1500)`, waiting until pending fetch/XHR calls hit zero, pending timers hit zero, and the DOM has been quiet for at least 250ms. So when this scenario goes red, it is because the behaviour is missing, not because the assertion fired while an AJAX call was still in flight. An honest red is what makes green mean something.

Recipe 702 Triangulation (Beck)

Idea

When the simplest implementation that passes one test is “return a hard-coded value”, you don’t yet have enough tests. Write a second test that forces the implementation to generalise. The technique is **triangulation**, two examples pin down the behaviour where one was ambiguous.

Webship-js translation

Every team has shipped the hard-coded version of something at least once, the discount that was literally `total - 10` because the only example used a $100 cart. The test stayed green for a year. The first $50 cart in production produced a negative total, and the postmortem asked how the tests missed it. They didn’t miss it. There was only one example, and one example cannot pin down a rule.

```
Scenario: 10% coupon discounts a $100 cart by $10
  ...

Scenario: 10% coupon discounts a $200 cart by $20
  ...
```

The first scenario alone could be satisfied by “always subtract $10.” The second forces the implementation to multiply by the cart total. Two examples, one inferred rule.

In Gherkin, the cheap way to triangulate is a `Scenario Outline` with an `Examples` table, one structure, several rows, each row a fresh chance to expose a hard-coded answer. Choose the rows deliberately: $100 and $100.00 exercise the same code path and triangulate nothing. $100 and $237.50 force the real multiplication into existence.

Recipe

When a test is too easy to pass with a hard-coded answer, add a second scenario with a different input, different enough that a faked implementation would fail it. The pair triangulates the behaviour.

Recipe 703 BDD as Translation (North, 2006)

Idea

Dan North coined BDD because he kept finding developers stuck on the word “test.” If you write “test something”, you fixate on the implementation; if you write “describe a behaviour”, you stay at the user-language level. BDD’s contribution was not a tool but a vocabulary: *given*, *when*, *then*; *features*, *scenarios*, *examples*.

Webship-js translation

You can spot a suite that lost this translation in one scroll: scenarios full of POSTs, payloads, and CSS classes that the product owner stopped reading months ago. Once that happens, sign-off becomes an act of trust, which means it is not really sign-off. Every feature file should be the vocabulary in action:

```
Feature: Coupon discounts
  As a shopper
  I want to enter a coupon code
  So that I get the advertised discount

  Scenario: Apply a coupon
    Given I am on "/cart"
    When I fill in "code" with "SAVE10"
    Then eventually I should see "Discount applied" within 2 seconds
```

The header section is North's "user story", role, action, value. The scenario is the example that proves the story.

Webship-js bakes the same translation into its step engine. `When I fill in "code" with "SAVE10"` resolves "code" label-first, the step looks for the visible label before falling back to id, name, or placeholder. You write the word the shopper sees on the screen; the step finds the input behind it. That is North's idea running at selector depth: the scenario speaks user, the framework speaks DOM, and the translation happens in the one place equipped to do it.

Recipe

Keep the user-language tier in feature files. Never let implementation vocabulary leak in. If you write `When I post to /api/coupons/apply`, you've lost the user lens. Rewrite as `When I click "Apply"`.

Recipe 704 The Ubiquitous Language (Evans, 2003; North, 2006)

Idea

Eric Evans introduced “ubiquitous language” in *Domain-Driven Design*: the team should use the same words in conversation, documentation, and code. Dan North made BDD a vehicle for that language, every Gherkin step uses domain terms, not implementation jargon.

Webship-js translation

The drift is slow and nobody decides it. Marketing renames “customer” to “member” in March. The UI follows in April. By June your steps still say customer, the product owner’s criteria say member, and every refinement session starts with two minutes of translation. Same system, two dialects. Your custom steps and feature files should use the vocabulary stakeholders actually use:

```
# Good (domain language)
When the customer applies the coupon "SAVE10"
Then the order summary should show a "$10.00" discount

# Bad (implementation language)
When the user POSTs to /api/orders/42/coupons
Then the response body should have property "discount":1000
```

The language layer goes deeper than feature files. Webship-js keeps a named-selector registry, JSON files in `tests/selectors/` mapping a human name to a CSS or XPath selector, auto-loaded through `worldParameters.selectors.files`. Name those entries in domain terms too (“order summary”, “discount banner”), and the ubiquitous language extends down into the one place implementation detail is allowed to live.

At scale: across twenty projects, vocabulary drift multiplies into a review tax, a QA lead moving between suites has to relearn what each team calls the same concept. Keep one shared glossary for cross-project terms and let each project add domain words, not synonyms. When a shared step li-

brary exists, the glossary is effectively its API documentation.

Recipe

Audit your step library quarterly for terms that drifted away from the domain. “User”, “customer”, “shopper” should match what the product team says. If marketing rebrands “customer” to “member”, update the steps, and the selector names with them.

Recipe 705 Three Amigos (Adzic, 2011)

Idea

Gojko Adzic’s *Specification by Example* describes the **Three Amigos** meeting: a product person, a developer, and a tester (or QA, or designer) sit together and turn a user story into concrete examples. The examples become the acceptance scenarios. Conversation happens once; the scenarios encode the agreement.

Webship-js translation

The expensive version of skipping this meeting is the sprint demo. The developer shows the coupon flow, the product owner says “that’s not what I meant, it should apply before shipping is calculated, not after”, and three days of work plus the scenarios that faithfully tested the wrong behaviour go back into the sprint. Thirty minutes of conversation would have caught it on a whiteboard.

For every meaningful feature, hold a Three Amigos session. Outputs:

- 3-7 concrete scenarios in Gherkin
- A shared understanding of edge cases (zero-quantity coupon, expired coupon, currency conversion)
- A list of open questions for the product owner

The scenarios drive both development (developer's red-green-refactor) and verification (tester's coverage check). One artefact, three uses.

At scale: when one QA team supports many product teams, Three Amigos is what keeps acceptance criteria comparable across projects, same format, same concreteness, same idea of what counts as an example. Rotate which QA engineer attends which project's sessions, and the conventions travel with them. For a PMO, the session's output is the first reviewable evidence that a story was actually understood before anyone built it.

Recipe

Run Three Amigos for any feature larger than a one-line bug fix. Schedule 30 minutes. Bring a laptop and write the scenarios live. Save them in the repo as `.feature` files before anyone codes the implementation.

Recipe 706 Living Documentation (Adzic)

Idea

Adzic's most consequential framing: feature files are not tests, they are **living documentation**. They describe what the system does in the present tense. When a behaviour changes, you change the scenario; the documentation updates automatically.

A wiki page about how the coupon system works will drift from reality within months. A `coupons.feature` file that runs on every commit cannot drift, if it drifts, CI fails.

Webship-js translation

Every team has the wiki page. "How coupons work", last edited fourteen months ago, three renames behind the UI, still the first search hit for new hires. Documentation that nothing executes is documentation that nothing corrects.

Publish your `.feature` files where stakeholders can read them. Many teams render Gherkin features into a static site (Cucumber's reporting plugins do this). The result: a product specification that's also the test suite that's also the regression guard. One source of truth, three audiences.

Webship-js ships the publishing half: `bin/generate-reports.js` turns each run's JSON output into an HTML report, and a PDF when you need an artefact to attach to a release ticket, stamped with metadata like `--app-version` and `--env`. A report generated on every build is living documentation with a timestamp: this is what the system did, on this version, in this environment, at this hour.

At scale: publish every project's report to one shared location and you have answered the hardest portfolio question, "is release X safe?", with the same artefact for every product. A manager who cannot read Gherkin can read a green report with the app version in its header. Twenty wiki pages drift in twenty directions; twenty generated reports cannot.

Recipe

Treat feature files with the same review rigour as production code. Pull-request reviews for `.feature` files should look at language, clarity, completeness, the same way you'd review docs.

Recipe 707

The Imperative-Declarative Test (North; Cucumber Book)

Idea

An *imperative* test reads as a list of clicks: "click the login button, type the username, type the password, click submit." A *declarative* test reads as a behaviour: "the user logs in successfully." The Cucumber book recommends declarative scenarios with imperative custom-step implementations.

Webship-js translation

Watch a new hire's first PR. They need a logged-in user, find a feature file that does it click-by-click, and copy all five steps into their scenario, the fifteenth copy in the repo. Six months later the login form changes and someone edits fifteen files. That is the cost of imperative scenarios: every detail repeats everywhere it is needed.

```
# Imperative (avoid in scenarios)
Scenario: User logs in
  Given I am on "/user/login"
  When I fill in "username" with "alice"
  And I fill in "password" with "secret"
  And I click "Submit"
  Then I should see "Welcome"

# Declarative (prefer)
Scenario: User logs in
  When I log in as "alice"
  Then I should see "Welcome"
```

The declarative version reads like a spec. The imperative click-by-click belongs inside the custom step `When I log in as "X"`.

Webship-js's 411 built-in steps are deliberately imperative, fill, click, select, check, assert. They are the assembly language. Custom steps are the domain language you compile down to it. The discipline is keeping each tier pure: scenarios say *what* in product words; custom steps say *how* in built-ins.

At scale: declarative steps are what make a step library portable across a portfolio. `When I log in as "alice"` reads identically on every product, even when one login is a modal and another is a dedicated page, the per-project step definition absorbs the difference. Teams that share declarative vocabulary across twenty suites can review each other's scenarios in minutes; teams that share imperative click-lists share nothing.

Recipe

When a sequence of steps occurs more than three times in your feature files, declarativise it, extract the imperatives into a named custom step. The scenario reads better; the implementation hides where it should.

Recipe 708 The Cucumber Lineage (Hellesøy; Wynne)

Idea

Cucumber originated as RSpec's "Story Runner" in 2008 (Aslak Hellesøy). It became `cucumber-ruby`, then `cucumber-jvm`, then `cucumber-js`. Each port preserved the Gherkin syntax but adapted to its host language. The Cucumber Book (Wynne, Hellesøy, 2012; second edition 2017) codified the practice: features as documentation, steps as code, tags as configuration.

Webship-js translation

Webship-js builds on cucumber-js v10, the latest JavaScript port. Most of *The Cucumber Book*'s advice carries over verbatim: write declarative scenarios, share setup via Background, parameterise with Scenario Outlines, use tags for environment-aware runs.

The lineage is worth citing when someone on your team calls Gherkin "the new thing we're trying". The syntax has been earning its keep in production suites since 2008. The tools underneath kept changing, Ruby to JVM to JavaScript, Selenium to Playwright, but a feature file written in 2010 still reads exactly like one written today. That tells you which part was the durable idea: the language, not the runner.

Recipe

Read *The Cucumber Book* if you've never read it. Most of webship-js's patterns are direct descendants. The differences are tactical (web-first matchers, smart-waits, AI integration); the strategy is the same as it was in 2012.

Recipe 709 Walking Skeleton (Freeman & Pryce, 2009)

Idea

Growing Object-Oriented Software, Guided by Tests (GOOS) opens with the **walking skeleton**: the smallest possible end-to-end implementation that exercises every layer of the system. It's not feature-complete; it just shows that the layers talk to each other. The walking skeleton is the platform on which subsequent tests are built.

Webship-js translation

The alternative is the project where tests "will come later". Later arrives in week six; the first scenario fails for five unrelated infrastructure reasons at once, missing browser dependencies in CI, wrong base URL, a report directory that isn't writable, and the team concludes acceptance testing is flaky. It was not flaky. It was six weeks of unverified assumptions cashing in on the same afternoon.

For a new project, the walking skeleton is one passing scenario:

```
Feature: Walking skeleton

  Scenario: Homepage loads
    Given I am on "/"
    Then I should see "Welcome"
```

It tests almost nothing. But it proves: the test framework runs, the fixture server runs, the browser launches, the CI pipeline executes, the report uploads. Every other test builds on that foundation.

In webship-js terms, that one scenario proves more than it reads. The browser launched with your `BROWSER` and `HEADLESS` settings. `LAUNCH_URL` points at a server that actually answers. The smart-wait init scripts installed cleanly. The JSON report landed in `tests/reports/`. The CI runner has every system dependency Playwright needs. Each of those fails loudly and cheaply now, or mysteriously and expensively later, underneath your fiftieth scenario.

At scale: turn the skeleton into a template repository. A new project means clone, set `LAUNCH_URL`, pick the matching pipeline from the thirteen CI configs webship-js ships (GitHub Actions through Bamboo), and be green in CI inside an hour. Track that hour, time-to-first-green is the most honest onboarding metric a multi-project team has. When project number seven takes three days, your scaffold needs work, not your people.

Recipe

First scenario of a new project: write the walking skeleton, get it green in CI, then everything else.

Recipe 710 Outside-In vs Inside-Out (GOOS)

Idea

Two ways to drive TDD: from the outside (start with an acceptance test, work down to units) or from the inside (start with units, work up). GOOS makes the case for **outside-in**: the acceptance test forces you to design the system from the user's perspective, not the developer's.

Webship-js translation

Inside-out feels productive right up to the end. The API is built, the models are tested, and then the UI work starts and reveals the flow needs a field the API doesn't return. Outside-in surfaces that on day one, because the scenario describes the screen the user sees, and the screen is the contract.

Webship-js is the acceptance-test layer. Use it to **start** a feature:

1. Write the `.feature` file with the user-visible behaviour.
2. Run it. It fails because no UI exists yet.
3. Drop into your unit test framework. Write the inner units to support the UI.
4. Build the UI. Re-run the feature. Green.

The acceptance test is the steering wheel. Unit tests are the engine.

One refinement: make sure the scenario fails *for the right reason*. Run it at step 2 and read the failure, it should be a web-first assertion timing out after its 5-second poll because the element doesn't exist yet, not a connection refusal or a broken fixture. A red you understand is a steering wheel. A red you don't is just noise.

Recipe

For every new feature, write the webship-js scenario first. Use it as the "definition of done." The feature isn't done until the scenario is green.

Recipe 711

Test Doubles, Dummy, Fake, Stub, Spy, Mock (Meszaros; Fowler)

Idea

Gerard Meszaros's *xUnit Test Patterns* (2007) and Martin Fowler's "Mocks Aren't Stubs" (2007) named the family:

- **Dummy**, passed but not used (placeholder)
- **Fake**, working implementation with shortcuts (in-memory DB)
- **Stub**, returns canned answers
- **Spy**, records calls
- **Mock**, pre-programmed with expectations; fails if expectations aren't met

The five aren't synonyms. Each fits a different testing problem.

Webship-js translation

Teams blur these words daily, someone says "mock the API" and means stub it; someone else builds a true mock with expectations and wonders why it breaks on every refactor. The five names matter because each one fails differently when the system changes.

In acceptance tests, you primarily use **stubs** (`the URL "/api/x" returns the JSON: """ {"data":"..."}"""`) and **spies** (`I start recording network requests` + `Then a POST request to "/api/y" should have been made`). Fakes happen at the fixture-server level. Mocks are rare in browser tests, you usually want stubs and assertions, not pre-programmed expectations.

Under the hood, the stub steps register a `page.route()` interception with glob matching, a pattern like `**/api/x` catches the call wherever it originates, and answer from your canned JSON without the request ever leaving the browser. The spy steps record traffic as it happens so you can assert on it afterwards. The same family covers the network conditions themselves: `is blocked`, `is delayed by 2000 ms`, `the network is offline`, doubles for the wire, not just the responses.

At scale: keep canned responses in shared fixture files instead of docstrings duplicated across suites. When the payments API grows a field, you update one fixture set and twenty suites inherit it, instead of hunting stub payloads through twenty repos. Treat fixture-shape drift like any other dependency upgrade: deliberate, versioned, announced.

Recipe

Pick the right double for the situation. A stub answers “what does the page receive?” A spy answers “what does the page send?” A mock answers “did the page interact with this dependency exactly this way?” Most acceptance tests need the first two.

Recipe 712 Mockists vs Classicists (Fowler)

Idea

Fowler distinguished two TDD camps. **Classicists** prefer real collaborators when possible; they test outcomes. **Mockists** mock every collaborator; they test interactions. Each style produces different tests and different designs.

Webship-js translation

Acceptance tests are inherently *classical*: they use the real browser, the real DOM, often the real backend. Webship-js encourages the classical approach, mock only what you must (slow APIs, third-party services), keep the rest real.

Every `page.route()` stub you add converts a fact into an assumption, the test now proves the page works against your guess of the API, not against the API. Sometimes that trade is right: third-party services you can't control, error paths you can't trigger on demand. Make it consciously, one route at a time, instead of mocking the whole backend because it was slow one Tuesday.

Recipe

Don't over-mock acceptance tests. Each mock is a fidelity loss. If your test passes against mocks and fails against the real system, the test wasn't worth running. Prefer a real (or fake) backend over a fully-mocked one for acceptance.

Recipe 713 The Test Pyramid (Cohn, 2009)

Idea

Mike Cohn's *Succeeding with Agile* (2009) introduced the **test pyramid**: many fast unit tests at the base, fewer integration tests in the middle, a small number of slow end-to-end tests at the top. The shape is economic, end-to-end tests are slow and brittle, so you want only as many as necessary.

Webship-js translation

Acceptance tests sit at the top of the pyramid. They're slow (seconds, not milliseconds) and broad (one scenario covers many layers). You don't want hundreds of them per feature, you want the **right** dozen.

The economics compound in CI. A unit test costs milliseconds on one process. An acceptance scenario costs a browser, seconds of wall-clock, and runner minutes someone pays for. The pyramid was never an aesthetic, it is a budget allocation.

At scale: multiply by the portfolio. Forty suites running nightly at thirty minutes each is twenty hours of runner time, every night. The pyramid question stops being per-feature and becomes per-portfolio: which scenarios earn an every-commit run, which run nightly, which run weekly before a release? Tag tiers, `@smoke` versus the full set, are the pyramid expressed in Cucumber.

Recipe

For every feature, ask: how few acceptance scenarios cover the behaviour? Three to five is usually right. More than ten is a smell, you're probably testing implementation details that belong in unit tests.

Recipe 714 The Testing Trophy (Dodds, 2018)

Idea

Kent C. Dodds proposed the **testing trophy** as a modern revision of the pyramid: more integration tests, fewer pure units. The argument: unit tests on highly mocked code prove little; integration tests catch real bugs.

Webship-js translation

The trophy thinking pushes you to write more acceptance tests than the strict pyramid would suggest, because acceptance tests **are** integration tests. They exercise multiple layers in a real browser. What made that uneconomical for a decade was the flake tax, and that is exactly the tax webship-js's machinery removes: `smartSettle` waits for real quiet (pending requests at zero, pending timers at zero, DOM stable) instead of sleeping, and web-first assertions retry every 100ms for up to 5 seconds instead of failing on first look. The trophy is only viable when an extra test costs minutes to write and almost nothing to maintain. That is the bar the tooling has to clear.

Recipe

Don’t dogmatically follow the pyramid. If your unit tests are mostly testing your mocking framework, you need fewer of them and more acceptance tests. The webship-js auto-wait keeps the trophy economically viable.

Recipe 715 The Honeycomb (Schaffer, 2018)

Idea

Andre Schaffer’s **honeycomb** is yet another shape: a small core of integration tests, with a thin shell of unit tests around individual modules and a thin top of end-to-end tests. Used at Spotify and similar microservices shops.

Webship-js translation

For a microservices architecture, your webship-js suite tests the user-facing surface only; service-level integration tests live in each service’s own repo; pure unit tests cover algorithmic complexity. The shape is per-team and per-architecture, not universal.

If your guild meetings have devolved into pyramid-versus-trophy-versus-honeycomb debates, the shape has become a proxy war. The useful questions are concrete: where did the last five escaped bugs come from? Which layer would have caught them? Answer those and the shape draws itself.

Recipe

Don’t pick a shape because a famous engineer wrote about it. Pick the shape that matches your architecture. The webship-js layer is the same in all three shapes, it’s the top of the user-facing pile.

Recipe 716 Test as Specification (Beck; Adzic)

Idea

Both TDD and SbE treat the test as the spec. The test is not a check **after** development; it is the description **before** development. The implementation must satisfy the test.

Webship-js translation

You can tell which kind your suite is from the git history. When `.feature` files land in the same pull request as the implementation, or a sprint later, in a ticket called “add tests”, they are regression tests. Useful, but they only ever confirm what got built. When the `.feature` file lands first and the implementation PR is the thing that turns it green, the scenario did design work: it decided what done looks like before anyone spent three days building the wrong thing.

Webship-js makes spec-first cheap because its steps bind to what the user sees, not to what the developer builds. `When I fill in "code" with "SAVE10"` resolves the field label-first at runtime, you can write that line before the form exists, and it starts passing the day a field labelled “code” appears. The spec never has to wait for the implementation to become writable.

Recipe

Try writing the next scenario before its feature exists. Run it. Watch it fail. Build until it passes. The discipline is uncomfortable for a week; valuable for the rest of your career.

Recipe 717 Refactor Tests Too (Beck)

Idea

The test code is code. It deserves the same refactoring care as production code. Duplicated assertions, deep nesting, magic strings, all the smells of production code show up in test code too.

Webship-js translation

Nobody schedules the decay. It arrives one copied step at a time, until the suite that took minutes to extend takes an afternoon, and the person extending it is a new hire who copies the worst example in the repo because it was the first grep hit. Test code rots exactly the way production code rots, for exactly the same reason: it gets edited under deadline and cleaned never.

Refactor feature files:

- Extract repeated 3+ step blocks into custom steps (Chapter 50)
- Use `Background:` for shared setup within a feature
- Use Scenario Outlines for repeated structure with varying data
- Name custom steps with domain language (Recipe 687)

Refactor step definitions too:

- Extract repeated logic into helper functions
- Move fixture data into a `fixtures/` directory
- Split monolithic `custom.js` into per-domain files

At scale: refactoring twenty suites one at a time never happens, there is always a release closer than the cleanup. What works is refactoring the shared layer instead: move the duplicated step definitions into one package every suite imports, and a single refactor pays out across the whole portfolio at once. Each project keeps only its scenarios and a thin custom-step file, the part that genuinely differs.

Recipe

Schedule a test-suite refactor sprint at least once a quarter. Cost: 2-3 engineer days. Value: orders of magnitude in faster feature delivery and lower maintenance.

Recipe 718 Tell, Don’t Ask (in Tests) (Hunt & Thomas)

Idea

The Pragmatic Programmer’s “tell, don’t ask” principle applies to test setup. Instead of:

```
const cart = new Cart();
cart.setItems([...]);
cart.setDiscount(0.10);
cart.setShipping('express');
```

Write:

```
const cart =
aCart().withItems([...]).withDiscount(0.10).withExpressShipping().build();
```

Builders, factories, and DSLs make test setup readable.

Webship-js translation

Open your worst feature file and find the scenario with eleven setup steps before the one step that actually matters. Whoever reviews it has to run a browser in their head just to locate the behaviour under test, and whoever copies it (someone will) propagates all eleven lines. A custom step that wraps complex setup is the moral equivalent of Hunt and Thomas’s builder:

```
Given a cart with 3 items, a 10% discount, and express shipping
```

The implementation builds the cart; the scenario reads the intent. Inside that implementation you choose the fastest honest route to the state, drive the real UI when the journey itself is what you are proving, or seed the cart through the API when it is only scaffolding for the assertion that follows. The scenario reads identically either way, which means you can swap a slow UI setup for a fast API one later without touching a single feature file.

At scale: builder sentences are how setup vocabulary stays portable. `Given a cart with 3 items, a 10% discount, and express shipping` can mean a different commerce stack on every project, the sentence travels, the implementation stays local. Keep the grammar consistent across suites and reviewers stop needing project-specific knowledge just to read setup.

Recipe

For any test that needs more than 5 lines of setup, build a factory or a custom step. Setup that takes a screen of scrolling is impossible to read; setup that's one declarative sentence is.

Recipe 719 The Saff Squeeze (Beck; Saff)

Idea

When a failing test produces a confusing error, **inline progressively** until the failure is in a single line. Beck described David Saff's technique: copy the failing assertion to a new test; replace one variable at a time with its concrete value; rerun. Eventually the test fails in a way that names the bug.

Webship-js translation

The failure that calls for this is never the clear one. It is the scenario that dies on step nine, `When I complete checkout`, with a timeout, and `I complete checkout` is a custom step wrapping eleven built-ins wrapping a helper someone wrote two years ago. The error names a symptom four layers above its cause.

For a confusing acceptance failure:

1. Copy the failing scenario.
2. Replace abstract steps with their concrete equivalents one at a time.
3. Re-run after each substitution. The point where it starts failing tells you the layer of the bug.

Webship-js sharpens the squeeze. Tag the scratch copy `@screenshots` and you get a capture after every step, not just the failure, a flip-book of exactly where reality left the script. Drop `Then print current URL` between steps as you inline; it is the cheapest probe there is. And because the copy is disposable, you can be as crude as you like, it gets deleted the moment it has named the bug.

Recipe

When a scenario fails mysteriously, don't add console.logs. Saff-squeeze it. Inline one step at a time until the failure is unmistakable.

Recipe 720 The Single Responsibility Test (Beck; SOLID)

Idea

Each test should test one thing. A test that asserts five things produces five possible failure modes; understanding which one fired is extra work.

Webship-js translation

One scenario, one behaviour. Cucumber stops a scenario at its first failing step, so a five-assertion scenario fails serially: you fix the first break, re-run, and meet the second, a drip-feed of bad news that turns one debugging session into five, each with a fresh browser launch in between. The scenario name should answer "what behaviour does this prove?" in one sentence. If you can't, split.

Recipe

Audit your scenarios. Any scenario whose name is "User can do X and also Y" should be two scenarios.

Recipe 721 Property-Based Testing (Hughes; Hypothesis)

Idea

John Hughes's QuickCheck (1999) introduced property-based testing: instead of writing example-based tests, write *properties* that should always hold, and let the framework generate random inputs to look for counterexamples. Hypothesis (Python) and fast-check (JavaScript) brought the technique to mainstream languages.

Webship-js translation

Triangulation (Recipe 702) pins a rule with two examples a human chose. The uncomfortable truth: humans choose polite examples, round numbers, valid dates, ASCII names. The inputs that break systems are the ones nobody would think to type. Acceptance tests are inherently example-based, the Gherkin syntax is built around examples. But you can layer property tests at the API level:

```
const fc = require('fast-check');

it('coupon discount is always non-negative', () => {
  fc.assert(fc.property(
    fc.float({ min: 0, max: 1 }),
    fc.float({ min: 0, max: 10000 }),
    (rate, cart) => {
      const discount = calculateDiscount(rate, cart);
      return discount >= 0;
    }
  ));
});
```

The property runs hundreds of random inputs. The acceptance scenarios cover the canonical examples. The split is economic as much as philosophical: hundreds of runs through a real browser is hours, while hundreds of runs against a pure function is a second, put the randomness where itera-

tions are free. It is also about audience: a Gherkin example is a communication artefact stakeholders chose on purpose, and a random `{rate: 0.6631, cart: 8841.07}` communicates nothing to anyone.

Recipe

Use property tests at the layer below acceptance. Use example-based acceptance tests for the user-facing surface. Don't try to do property testing in Gherkin, it's the wrong layer.

Recipe 722 Mutation Testing (Osherove)

Idea

Roy Osherove's *The Art of Unit Testing* discusses **mutation testing**: a tool deliberately injects bugs into your code (changing `>` to `<`, removing `!`, etc.) and runs your tests. Bugs that no test catches mean your tests are inadequate, not just whether they pass, but whether they would notice a bug.

Webship-js translation

There is a special dread in discovering that a scenario which has passed four hundred runs in a row would also have passed with the feature deleted. Green tells you the assertions held. It never tells you the assertions were worth holding, that question needs a deliberate experiment.

Mutation testing is mainly for unit-test suites. For acceptance, the moral equivalent is "what happens if I delete this assertion?" If no other scenario catches the gap, the assertion is unique-value and worth keeping. If many other scenarios catch the gap, the assertion is redundant.

At scale: a delete-one-assertion pass across forty suites will never happen in one heroic week, rotate it. One suite per sprint, starting with the scenarios that guard each product's checkout-equivalent. Keep the output: a short list per suite of which scenarios are load-bearing and which are scaffolding. That list is also your priority order the next time the nightly run is over budget and something has to move to weekly.

Recipe

Once a year, do a “delete one assertion” pass across your suite. Anything that, when removed, leaves no test catching the regression is unique-value. Anything that, when removed, leaves five other tests catching the regression is fat.

Recipe 723 Tests as a Conversation (Adzic; Liz Keogh)

Idea

Liz Keogh argues a BDD scenario is the *artefact* of a conversation. The conversation matters more than the artefact. The Three Amigos meeting is where shared understanding happens; the scenarios are notes from that meeting.

Webship-js translation

A scenario written without the conversation is half-value. The same words can mean different things to different stakeholders, “discount applies to the order” means before shipping to the developer and after shipping to the finance manager, and both read the same Gherkin line and nod. The conversation forces the ambiguity into the open while it is still cheap to fix; the suite stays green either way, which is exactly why the suite alone cannot catch it.

Recipe

Don’t skip Three Amigos to “save time.” The 30 minutes you save become 5 hours of rework when the developer’s interpretation differs from product’s interpretation. Run the meeting. Write the scenarios in the meeting.

Recipe 724 Examples as Tests, Tests as Documentation (Adzic)

Idea

Adzic's circular insight: examples drive development (they're tests), and the same examples document the system (they're docs). The same artefact serves both. The implication: write examples that read well as documentation, not just as tests.

Webship-js translation

The honest test takes one minute: ask a product manager to read a scenario aloud. If their eyes glide over `Then the JSON response should have "/data/discount" equal to 1000`, your documentation has an audience of one team, the team that needs it least, because they wrote it. Re-read your scenarios from a non-engineer's perspective. If a product manager couldn't read the scenario and understand what the system does, the scenario is too implementation-laden.

At scale: the reading session doubles as a portfolio consistency check. When one product's scenarios say "the customer" and another's say "the end-user entity", everyone who moves between projects, QA leads, PMs, support, pays a translation tax on every visit. Cross-team reading sessions flush out the dialects, and the generated HTML reports give non-engineers a place to do the reading without cloning a repo.

Recipe

Hold a regular "scenario reading session" with product and design. They read the scenarios aloud. Anywhere they get confused or have to ask "what does this mean?", that's a scenario to refactor.

Recipe 725 The Walking-Backwards Mistake (Adzic)

Idea

A common mistake: write the implementation, then write the scenarios that match what the implementation does. The scenarios become a tautology, they pass by definition. Adzic calls this the “walking backwards” failure mode.

Webship-js translation

You can usually date the moment a suite started walking backwards: it is the sprint when “write the tests” became a separate ticket, scheduled after the feature ticket, often assigned to a different person. Whatever that ticket produces will describe the implementation, quirks included, because the implementation was the only specification its author had. Write scenarios **before** implementation, or at least independently of it. If you find yourself writing a scenario to fit an existing endpoint’s quirks, stop, the quirks are probably bugs, and you are about to give them a green checkmark that makes them permanent.

Recipe

A test that has never been red is a test you don’t trust. New scenarios should fail first.

Recipe 726 The Coverage Trap (Beck)

Idea

High test coverage is necessary but not sufficient. A suite with 95% coverage and weak assertions still ships bugs. Coverage measures lines executed, not behaviour verified.

Webship-js translation

The number arrives in a slide deck: “95% coverage.” Two sprints later a customer finds the bug the deck said could not exist, and the awkward retro rediscovers what coverage always measured, that lines ran, not that anyone checked the result of running them.

You can’t measure acceptance-test coverage in line-counts. The right metric is **behaviour coverage**: are all the documented user journeys exercised? Are all the documented edge cases exercised? Coverage tools won’t tell you. You and your team will, by reading the scenarios.

At scale: behaviour coverage is also the only coverage a PMO can compare across products. Line coverage of a CMS site and a React SPA measure different universes; “every documented checkout journey has a green scenario” means the same thing on both. Keep the journey list next to each suite and report against it, it turns “is release X safe?” from a feeling into a checklist anyone can audit.

Recipe

Don’t chase a line-coverage number for acceptance tests. Track behavioural coverage by listing user journeys and matching them to scenarios. A journey without a scenario is a gap; a scenario without a journey is fat.

Recipe 727 Tests as a Safety Net (Beck)

Idea

The deepest reason to write tests is courage. Tests let you change code without fear. A suite that catches regressions in 30 seconds turns “should we refactor?” into “yes, let’s go.” A suite that takes 4 hours to run does the opposite.

Webship-js translation

Watch what happens when someone proposes replacing the auth module. On the team with the four-hour suite, the room goes quiet and the refactor dies in the backlog. On the team with the fifteen-minute suite, somebody says “the scenarios will catch it” and the work starts Monday. Same risk, different courage, and the only variable is the feedback loop.

A fast, reliable webship-js suite is courage. It says “yes, let’s rip out the old auth system” because you’ll know in 5 minutes if anything important broke.

At scale: courage is set by the slowest suite a change must clear. If a platform upgrade touches twenty sites and three of them have 90-minute suites, those three set the pace for all twenty, so the upgrade gets batched, postponed, and riskier. Put wall-clock time per suite on one dashboard and treat a suite drifting past budget as a defect to fix, not a fact of life.

Recipe

Track your suite’s wall-clock time. Anything over 15 minutes for the full suite breaks the courage loop. Shard, parallelise, or split into smoke-and-full tiers (Chapter 47, Recipe 532) to keep iteration tight.

Recipe 728 Don’t Test the Framework (Fowler; Kent Beck)

Idea

A common waste: writing tests that verify the framework’s own behaviour rather than your code. If Cucumber’s `Given X` step works, you don’t need a test that asserts `Given X` worked, the framework’s tests already do.

Webship-js translation

These scenarios accumulate innocently. Someone wants a “quick sanity check”, writes a scenario that navigates and then asserts the navigation happened, and it passes forever, because what it actually proves is that Playwright works. Three years later there are forty of them, each burning a browser session per run and guarding nothing the framework’s own test suite doesn’t already guarantee.

```
# Wasted test -- verifies Playwright/Cucumber, not your app
Scenario: I can navigate to the homepage
  Given I am on "/"
  Then the url should match "/"

# Real test -- verifies your app's contract
Scenario: Homepage shows the welcome message
  Given I am on "/"
  Then I should see "Welcome to Webship.co"
```

At scale: dead-weight scenarios are a CI line item. One asserts nothing app-specific yet still costs a browser launch, a settle cycle, and runner seconds, multiplied by forty suites and 365 nights. A yearly hunt for assertions that would pass on any page is the cheapest CI cost cut available, because deleting them loses exactly nothing.

Recipe

Re-read every scenario. If it would pass against any random page, it’s testing the framework. Replace with a real assertion.

Recipe 729 The Right Failure Message (Beck)

Idea

A failing test should tell you what's wrong in one sentence. "Expected true but got false" is useless. "Expected the order total to be $100 but got $90" is actionable. Beck spent years iterating xUnit's failure messages because the message **is** the test's interface to a debugging engineer.

Webship-js translation

The message gets read at the worst possible moment, 7:40am, twenty red scenarios, release at noon, by someone who did not write the test. "Assertion failed" at that moment is not a message; it is homework.

Webship-js's friendly-error pipeline (Chapter 44) is the same idea applied to acceptance tests. The built-in assertions already speak in specifics, expected versus actual, selector and timeout named, and every failure arrives with its screenshot (capture-on-failure is on by default) plus whatever console errors the JS-error hooks collected. Your custom steps are the one place vagueness can creep back in. Hold them to the same standard:

```
throw friendly(`Expected the order total to be $${expected} but got
$${actual}. The page is ${url}, the cart contains ${itemCount} items.`);
```

At scale: failure messages are the interface between your projects and whoever is on triage rota. When the person reading the morning's failures supports twelve products, a message that names the expected value, the actual value, and the URL is the difference between thirty seconds and thirty minutes per red scenario. Write every message for a reader who has never seen the project, on a multi-project team, that reader is the normal case.

Recipe

Audit your custom-step error messages once a quarter. Anywhere the message is "Assertion failed", rewrite to include the values being compared and the context the user would need.

Recipe 730 One Failure, One Diagnosis (Beck)

Idea

When a single test failure produces multiple stack traces, the cause is multiple. Aim for tests where **one failure means one cause**. The classical TDD discipline, one assertion per test, when reasonable, enforces this.

Webship-js translation

Broad scenarios feel efficient on the day you write them, one browser session, ten assertions, look at all that coverage. They bill you later, on the morning one goes red and the name "Checkout works correctly" tells you nothing about which of its ten promises broke. A scenario with 10 `Then` steps will produce 10 possible failures. The first to fail stops the rest, so a single red run shows you one break and hides the other nine behind it. Prefer narrow scenarios over broad ones.

Recipe

When a scenario has more than 5 `Then`s, look for natural splits. Two narrower scenarios beat one wide one for diagnosability.

Recipe 731 The First Test Hardest (Beck)

Idea

Writing the first test for a new codebase is the hardest because you have to decide on the fixture, the harness, the build, the CI. Once it's in place, the second test takes 10% of the effort. Get the first one in early, even if it's almost-trivial.

Webship-js translation

Teams postpone the first test waiting to feel ready, the naming-convention debate, the folder-structure debate, the should-we-use-tags debate. Beck's point cuts through all of it: the first test's job is to exist, because it forces every infrastructure decision (fixture, browser, CI, report) to be made once, cheaply, while nothing depends on them yet. The walking skeleton (Recipe 709) is the first test. Get it green in CI before you've decided on naming conventions, before you've written your style guide, before you've designed your test World. Everything else follows.

Recipe

First commit on a new project should include a passing `.feature` file. Even if it's `Scenario: Homepage exists`. The cost is 10 minutes; the value is unlocking every subsequent test.

Recipe 732 Tests Are an Honest Record (Beck)

Idea

If a test passes, the system has the behaviour the test describes. If a test fails, it doesn't. There is no in-between. The test is honest in a way intuition isn't. "I'm pretty sure the login works" is a feeling. "The login scenario is green" is a fact.

Webship-js translation

This cuts both ways, and the painful direction is the second one. "I'm sure that still works" has shipped more regressions than any other sentence in software, and its mirror, "that test always fails, ignore it", is how a team trains itself to stop reading its own record. Believe the suite. If a scenario says the system does X and you think it doesn't, either the scenario is wrong (refactor) or your belief is wrong (the system actually does X). Don't override the suite with intuition, and don't leave a scenario in the suite once you've decided its word no longer counts.

Recipe

When a scenario contradicts your intuition, investigate the scenario, not your intuition. Often the scenario is right and you've been operating on stale beliefs.

Recipe 733 Refactoring Without Tests Is Editing (Feathers; Fowler)

Idea

Michael Feathers's *Working Effectively with Legacy Code* defines legacy code as "code without tests." Refactoring legacy code without first writing characterisation tests isn't refactoring, it's editing with hope.

Webship-js translation

The scary version is the rewrite nobody can verify. The old checkout "just works", the person who built it left in 2023, and now you are replacing the framework underneath it. Without tests, every behaviour you didn't know existed is a behaviour you are about to delete, and you will learn the full list from support tickets.

Before refactoring a UI feature, write the acceptance scenario for the **current** behaviour. Run it. Green. Then refactor. The scenario tells you when you've broken something.

At scale: characterisation suites are how a platform team survives portfolio-wide upgrades. When we moved Varbase through major Drupal upgrades at Vardot, the acceptance suite was the characterisation suite, the same scenarios ran against the old major and the new one, and every difference was either a planned change or a regression caught before a single site upgraded. One suite, written once, de-risked the same migration for every site built on the distribution.

Recipe

For any feature you're about to rewrite, write the acceptance scenarios first as a characterisation suite. Spend a day on it. The day pays back tenfold during the rewrite.

Recipe 734 The Test Coverage / Behaviour Coverage Distinction (Adzic)

Idea

Two different coverage measures:

- **Test coverage**, what lines of code execute
- **Behaviour coverage**, what user behaviours are exercised

A 95%-test-coverage suite that hits 30% behaviour coverage is dangerous. The reverse, 60% test coverage but 95% behaviour coverage, is much safer.

Webship-js translation

The two numbers answer different questions from different people. Test coverage answers the engineer's question, did my code run under test? Behaviour coverage answers the product owner's, can a customer actually do the things we sold? A release decision made on the first number alone is a decision made about the wrong question. Webship-js scenarios measure behaviour coverage. Track them by category:

- Happy paths covered? (yes / no)
- Common error paths covered?
- Edge cases (zero, negative, max) covered?
- Cross-locale / cross-device covered?

Recipe

Maintain a behavioural-coverage checklist alongside your test suite. The checklist tells you what's tested and, more importantly, what's not.

Recipe 735 BDD is About Communication (North; Adzic; Keogh)

Idea

BDD's core idea is not "tools" or "syntax", it's **communication**. Scenarios are a contract among engineering, product, and QA. The shared understanding is the value; the executable test is a happy side effect.

Webship-js translation

The tooling makes this easy to forget. A team can install cucumber-js, write five hundred scenarios, keep them green in CI, and still practice zero BDD, because nobody outside engineering ever read one. Treat scenarios as the contract. If product reads a scenario and disagrees, that's the highest-value moment in the whole delivery cycle to catch a misunderstanding, the code does not exist yet, so the fix costs an edit to a text file instead of a sprint.

Recipe

Add "review the scenarios" as an explicit step in your product workflow. Product manager signs off on the scenarios; engineering builds against them.

Recipe 736 Don’t Skip the Refactor (Beck)

Idea

The “refactor” step in red-green-refactor is the one teams skip first. The temptation: once the test is green, move on. The cost: technical debt compounds, tests become hard to read, the suite gets slow.

Webship-js translation

The skip never announces itself. It sounds like “we’ll clean this up next sprint”, said sincerely, sprint after sprint, until the suite is a hundred scenarios of copy-paste that everyone resents extending. After each new scenario passes, allocate 5-10 minutes to refactor while the context is still in your head:

- Is the scenario name still accurate?
- Are there duplicated steps that should be a `Background:`?
- Are there imperative steps that should be declarative?
- Are there custom steps to extract?

Recipe

Build “refactor minute” into your team’s TDD loop. Some teams use a literal kitchen timer set to 5 minutes after a green light. The discipline pays off.

Recipe 737 Test Names as Sentences (Beck)

Idea

A good test name reads as an English sentence describing the behaviour. `test_foo_bar_baz_1` is uninformative. `When a coupon code expires, applying it shows the "expired" message` is.

Webship-js translation

Gherkin scenarios are named sentences by design:

```
Scenario: Expired coupons show an "expired" message when applied
```

Use scenario names as the index into your suite. A grep through the names should produce a readable product overview. Names are also what every report leads with: the HTML report renders one line per scenario, and a manager scanning it reads names, never steps. “Coupon test 3” wastes that audience; “Expired coupons show an expired message when applied” briefs them.

At scale: consistent naming turns grep into a portfolio query. When all twenty suites name scenarios as behaviour sentences, one search across repos produces a readable inventory of what the whole product line actually does, the cheapest capability audit a PMO will ever get. The convention costs nothing on day one; retrofitting it three years in costs a quarter.

Recipe

After writing each scenario, re-read just its name. If it reads as a sentence describing a real behaviour, ship. If it reads as a label or a category, rename.

Recipe 738 Beyond Green and Red, The Pending Test (North; Cucumber Book)

Idea

A test can be **pending**, written, but not yet implementing. Useful when you've identified a behaviour during a Three Amigos session but don't have time to implement immediately.

Webship-js translation

Every good Three Amigos session produces more scenarios than the sprint can absorb, and the overflow has to live somewhere. In most teams it lives in meeting notes, which is to say, it dies. The fifth time someone says "we said we'd test that" after a production incident is when teams learn to keep the overflow in the repo instead.

Tag scenarios `@pending` and exclude them from the default run with a tag expression (`--tags "not @pending"` in your default profile):

```
@pending
Scenario: Loyalty points award on first purchase
  Given I am a new customer
  When I place my first order
  Then I should see "You earned 50 loyalty points"
```

The scenario captures the requirement now; implementation happens later.

At scale: @pending is backlog evidence a PMO can audit. Counting pending scenarios per project shows where agreed behaviour is outrunning implementation, a leading indicator no ticket system gives you, because tickets rarely state acceptance criteria in executable form. A project whose pending count only ever grows is holding conversations it cannot cash.

Recipe

Use `@pending` instead of TODO comments. A scenario tagged @pending is searchable, tracked, and visible in reports as outstanding work.

Recipe 739 The Definition of Done (Adzic; Crispin & Gregory)

Idea

Lisa Crispin and Janet Gregory's *Agile Testing* defines done as "feature complete AND tests passing AND examples documented." Acceptance scenarios are the documentation. A feature isn't done until its scenarios are green.

Webship-js translation

Without a definition, "done" decays into "demoed". The feature works in the sprint review, the ticket closes, and three weeks later the edge cases nobody wrote down surface in production, where edge cases prefer to surface. Adopt the Crispin-Gregory definition in your team's workflow:

- Feature merged → required scenarios present
- Feature deployed → all scenarios green in production environment
- Sprint complete → no `@pending` from this sprint

At scale: a shared definition is what makes "done" comparable across a portfolio. When every project's gate is "relevant scenarios green in CI", release readiness review becomes twenty report links instead of twenty status meetings, stamp each report with `--app-version` and `--env` and the evidence describes itself. The projects that negotiate private definitions of done are the ones that surprise you in the release retrospective.

Recipe

Make the definition explicit. Put it on the wall, the wiki, the PR template. “DoD: relevant webship-js scenarios are green in CI.”

Recipe 740 Acceptance Tests Are Slow on Purpose (Cohn; Fowler)

Idea

Acceptance tests are slower than unit tests because they exercise more layers. That’s not a bug, it’s how they catch integration issues. Trying to speed them up to unit-test levels usually means mocking out the layers they’re meant to test.

Webship-js translation

Sooner or later someone proposes “speeding up the suite” by stubbing the backend, and the proposal sounds like engineering. Recipe 712 named the price: every stub trades a fact for an assumption. The seconds an acceptance scenario spends are not waste, they are real fetches, real renders, real settles. `smartSettle` waiting for the page’s pending requests and timers to reach zero *is* the test; that wait is precisely the integration the unit tests below could never check.

A 5-minute webship-js suite is **fast** for acceptance tests. A 30-second suite is suspect, did you mock the backend? If so, you may have lost the value.

Recipe

Optimise acceptance suite speed by parallelisation (Chapter 47, Recipe 551) and selective tagging (smoke vs full). Don’t optimise by mocking out the system you’re trying to test.

Recipe 741 Failure as Feedback (Beck)

Idea

A test failure is feedback. The faster the feedback, the more often you act on it. Beck argues for feedback in seconds, not minutes, because a minute of feedback breaks flow.

Webship-js translation

Feedback that arrives in seconds changes your next keystroke; feedback that arrives tomorrow changes your calendar. Acceptance tests will never hit Beck's seconds, but the distance between "thirty seconds after save" and "after the nightly run" is the distance between fixing a bug while the code is still in your head and reconstructing the whole context at standup.

Run a tagged smoke subset on every save (with a file-watcher). Run the full suite in CI on every commit. Receive feedback every 30 seconds locally, every 5 minutes in CI.

Recipe

Cucumber-js has no watch mode of its own, so wrap it in one: `npx nodemon --ext js,feature --watch tests --exec "npx cucumber-js --tags @critical-path"`. Save a file, smoke runs, get told immediately if you broke something. The 30-second feedback loop is the closest acceptance testing gets to TDD's tightness.

Recipe 742 The Test Suite as Product (Adzic)

Idea

The test suite is a product the engineering team builds, maintains, and uses. It has users (the team), features (which behaviours it covers), bugs (flakes, false negatives), and a roadmap (which areas need more coverage). Treat it like a product.

Webship-js translation

Suites nobody owns develop the symptoms of any unowned product: features nobody asked for, bugs nobody fixes, users who quietly stop trusting it. The day a developer mutters “just re-run it, it does that sometimes” is the day your product lost a customer.

Track the suite’s health metrics:

- Pass rate (target: 100% on `main`)
- Flake rate (target: <1%)
- Wall-clock time (target: <15 minutes)
- Behaviour coverage (target: tracked, growing)
- Custom-step count (target: small, audited)

The raw material is already there: every run writes `tests/reports/cucumber_report.json`, and the report generator stamps each HTML or PDF artefact with `--app-version`, `--env`, `--browser`, and any `--metadata Key=Value` you pass. Pass rate, duration, and flake trends are a small script over JSON you are already producing.

At scale: one suite’s metrics are a curiosity; forty suites’ metrics on one board are an operations tool. Normalise the metadata, same environment names, same version stamps, and the board starts answering real questions: which products flake worst, which suites blew their runtime budget, which have not gained a scenario in six months while features kept shipping. That last one is the silent killer: the suite didn’t break, it just stopped describing the product.

Recipe

Add suite-health to your engineering metrics dashboard. The suite is too important to be an afterthought.

Recipe 743 Tests as Designer (Beck)

Idea

Writing the test first **forces** good design. If your test is painful to write, the code is painful to use. The test is a design tool.

Webship-js translation

At the acceptance level, a scenario that's painful to write usually means the UI is hard to use. Eight steps for a single user action suggests too many clicks; deeply nested step dependencies suggest tangled UI. The signal is easy to dismiss as test-framework friction, "webship-js needs too many steps for this", when it is actually the interface confessing. If QA needs twelve steps to apply a coupon, so does every customer, and customers don't get a green checkmark for finishing.

Recipe

Use the scenario's readability as a UI signal. "I had to write 12 steps for this simple action" is feedback for the designer.

Recipe 744 The Discipline of Small Steps (Beck)

Idea

Beck's *TDD by Example* opens with the smallest possible code change between red and green. Why? Because small steps mean you always know what you just did. If the test goes red after the change, the bug is in the last 30 seconds of code, not in the last 30 minutes.

Webship-js translation

Implementing a feature scenario? Make the smallest change that moves the scenario closer to green. Run it. Move on. The scenario doubles as a progress bar: each run should fail one step further down than the last, and a failure that jumps *backwards* tells you the latest change broke something earlier, while that change is still thirty seconds old and trivially revertible.

Recipe

Resist the urge to "implement everything for this scenario in one go." Each commit should change one thing. The scenario tells you when you're done.

Recipe 745 The Two Heuristics for Test Names (Beck; Fowler)

Heuristic 1: Behaviour, Not Implementation

Bad: `Scenario: GET /api/users returns 200` **Good:** `Scenario: Anyone can list public user profiles`

Heuristic 2: Specific, Not Generic

Bad: `Scenario: User login` **Good:** `Scenario: User logs in with valid credentials` **Better:** `Scenario: User logs in successfully and lands on the dashboard`

Webship-js translation

Both apply directly to your scenario names. Read each name aloud. If it sounds like a category label, replace with a behaviour sentence.

Recipe

Hold a scenario-naming review. Read each name. Anything that fails either heuristic, rename.

Recipe 746 The Edge Case Hunt (Cucumber Book)

Idea

Wynne and Hellesøy recommend a deliberate edge-case hunt during Three Amigos. For each user story, ask:

- What if the input is zero, negative, very large, or empty?
- What if the user is offline, on a slow network, or using an old browser?
- What if two users do this simultaneously?
- What if the operation is repeated?

Each “what if” that produces interesting behaviour becomes a scenario.

Webship-js translation

Add an edge-case section to your `.feature` files:

```
Feature: Apply a coupon

  # Happy path
  Scenario: Valid coupon applies
    ...

  # Edge cases
  Scenario: Empty coupon code shows validation error
  Scenario: Coupon applied twice does not double-apply
  Scenario: Expired coupon shows expired message
  Scenario: Coupon larger than cart total caps at cart total
```

Recipe

For every new feature, write at least three edge-case scenarios alongside the happy path. The edge cases are where the bugs live.

Recipe 747 The Definition of Ready (Crispin & Gregory)

Idea

A user story is **ready** for development when it has:

- A clear acceptance criteria
- A small enough scope to fit in a sprint
- Concrete examples (your scenarios)

Stories without acceptance scenarios are not ready. They're wish-lists.

Webship-js translation

Before estimating or scheduling a story, write its acceptance scenarios. If you can't, the story isn't well-enough understood to estimate. Send it back to product for refinement.

Recipe

Add scenarios as a Definition of Ready in your team's process. The 30 minutes it takes to write three scenarios catches half a sprint's worth of ambiguity.

Recipe 748 The Production Test (Crispin & Gregory)

Idea

Some tests should run in **production**, against the real live system, to catch incidents the moment they happen. Synthetic monitoring, a probe that does a real user journey every 5 minutes, is a form of acceptance test in production.

Webship-js translation

Tag a subset of scenarios `@production-safe` (read-only, idempotent, low-traffic). Schedule them to run against production every 5-15 minutes. Alert on failure.

Recipe

Pick three smoke scenarios. Run them against production every 10 minutes. The 5-second alert when checkout breaks at 3am beats the 1-hour delay from waiting for a customer to report it.

Recipe 749 Test Heuristics (James Bach; Lisa Crispin)

Idea

Software testers maintain heuristics, rules of thumb for finding bugs. Examples: “test at boundaries”, “test for races”, “test the error path more than the happy path.” Lisa Crispin’s *Agile Testing* lists dozens.

Webship-js translation

For each new feature, run a heuristic checklist:

- Boundary: zero, one, many, max
- Race: two clicks rapid, two sessions concurrent
- Network: slow, offline, intermittent
- Time: midnight rollover, leap second, DST transition
- Locale: RTL language, multi-byte characters, currency
- Failure injection: backend returns 500, 404, timeout

Recipe

Build a personal or team heuristics checklist. Run it for every feature. The list grows as you encounter new bug-classes.

Recipe 750 The Long Discipline (Beck; Adzic; Wynne)

Idea

Every author in the canon ends with the same point: testing is a discipline. Not a tool, not a methodology, a discipline. Like any discipline, the gains compound over years. The team that practiced TDD or BDD for ten years writes faster, cleaner code than the team that decided to “try TDD” for a quarter.

Webship-js translation

Don’t try webship-js for a quarter. Adopt it for a decade. Build the muscle of writing scenarios first. Build the habit of refactoring tests. Build the team norm of Three Amigos. The compounding starts slow and ends powerful.

Recipe

Pick one practice from this chapter. One. Practice it for a month. Then pick another. After a year, you'll have twelve practices internalised, and a test suite that compounds value every commit.

Closing Thoughts From the Canon

A short list of quotes worth reading in full, with the verbatim words from each author's own book:

- **Kent Beck**, *Test-Driven Development by Example* (2002): "Clean code that works." (Quoted in Harry Percival's *Test-Driven Development with Python* as the goal of TDD.)
- **Dan North**, "Introducing BDD" (2006); foreword to *The Cucumber Book* (2012): "BDD has come a long way since I first started talking about it in 2003… I was simply trying to find better ways to explain the revelatory practice of TDD… Over time we were able to take the action closer to the business stakeholders and describe the multilayered approach that is modern BDD (and is also, ironically, classic TDD, which Kent Beck described from the very outset as working on multiple levels of abstraction)."
- **Matt Wynne & Aslak Hellesøy**, *The Cucumber Book* (2012, 2017): "Cucumber is a friendly tool. It wants to be part of your team… Cucumber will volunteer to do all the boring repetitive checks… freeing your testers up to do interesting, creative work instead." And: "Cucumber is more than a testing tool; it's a collaboration tool."
- **Ian Dees, Matt Wynne, Aslak Hellesøy**, *Cucumber Recipes* (2013): "Cucumber is first and foremost a communication tool. It helps you express in clear terms what your software is supposed to do and why."
- **Steve Freeman & Nat Pryce**, *Growing Object-Oriented Software, Guided by Tests* (2009): "Writing a test first forces us to clarify our intentions, and we don't start the next piece of work until we have an unambiguous description of what it should do. The process of writing a test first helps us see when a design is too rigid or unfocused."
- **Robert C. Martin**, foreword to *The RSpec Book* (2010): "The meme is *Craftsmanship*."
- **Harry Percival**, *Test-Driven Development with Python* (2014): "Obey the Testing Goat", the discipline of writing tests first, even when deadlines whisper otherwise.

- **Enrique Amodeo**, *Learning Behavior-Driven Development with JavaScript* (2015): “A lot of people see the whole TDD/BDD approach as a black-and-white decision. Either you do not do it, or you try to achieve a hundred percent test coverage. The real world calls for a more pragmatic approach: write the tests that really pay off and do not write those that do not give you much value.”
- **Gojko Adzic**, *Specification by Example* (2011) and *Bridging the Communication Gap* (2009): living documentation, three amigos, key examples.
- **Liz Keogh**, BDD talks (2008-present): “BDD is about the conversation, not about the tools.”

If this chapter is the only thing you read on BDD theory, you’re well-armed. If it’s the bibliography you read **before** the books, you’ll find familiar ideas wherever you go. Either way, you stand on the shoulders of a generation that worked these patterns out by hand. Build on their work; don’t reinvent it.

Appendices

Appendix A: Why Playwright Won, The Journey from Nightwatch.js

I need to be honest with you about something. webship-js did not start with Playwright. The first version, webship-js 1.0.x, was built on top of Nightwatch.js. And for a while, it worked. It got the job done. Teams used it, tests ran, bugs got caught.

But as web applications grew more complex, SPAs, dynamic content, client-side routing, WebSocket connections, lazy loading, iframes inside modals inside iframes, Nightwatch.js started showing cracks. Not because it was a bad tool. It was a good tool for its time. But the web evolved, and the testing tools needed to evolve with it.

This appendix is the story of that evolution. If you are coming from Nightwatch.js, Selenium, or Cypress, this will help you understand why Playwright is the foundation of webship-js 2.0, and why it matters for your test suite.

The Testing Tool Landscape

Let me lay it out. Here is the landscape as it stands today:

The Testing Tool Landscape (2024-2025)

Generation 1: Selenium-Based

Selenium WebDriver (2004)	The pioneer. Still alive.
`Nightwatch.js (2014)`	`Node.js + Selenium. Simple.`
WebdriverIO (2012)	Full-featured Selenium wrap.
Protractor (2013)	Angular-focused. Deprecated.

Generation 2: Non-Selenium

Cypress (2017)	In-browser execution. Fast.
TestCafe (2016)	Proxy-based. No WebDriver.
Puppeteer (2017)	Chrome DevTools Protocol.

Generation 3: Modern Multi-Browser

Playwright (2020)	CDP + Firefox + WebKit.
Built by the team who made Puppeteer.	
This is where the industry is heading.	

FIGURE A.1 *The Testing Tool Landscape*

Each generation solved problems the previous one could not. Selenium gave us browser automation. Nightwatch.js made Selenium accessible to Node.js developers. Cypress made tests fast and developer-friendly. Playwright took everything that worked and fixed everything that did not.

Nightwatch.js: What It Did Well

I do not want to bash Nightwatch.js. That would be unfair. Let me give credit where it is due.

Nightwatch.js made browser testing approachable for Node.js developers at a time when Selenium felt like configuring a spaceship. You installed it, pointed it at a browser, and wrote tests in JavaScript. The API was clean:

```
// Nightwatch.js style
module.exports = {
  'Login test': function (browser) {
    browser
      .url('http://localhost:3000/login')
      .waitForElementVisible('#email', 5000)
      .setValue('#email', 'user@example.com')
      .setValue('#password', 'secret123')
      .click('#login-button')
      .waitForElementVisible('.dashboard', 5000)
      .assert.containsText('.welcome', 'Welcome')
      .end();
  }
};
```

That is readable. That is straightforward. For a traditional multi-page application with server-rendered HTML, this worked perfectly fine.

Nightwatch.js strengths:

- **Simple setup** for its era, npm install and go
- **Clean chaining API**, fluent, readable test code
- **Built-in assertions**, .assert and .verify out of the box
- **Page Objects**, first-class support for the Page Object pattern
- **Selenium ecosystem**, access to the entire Selenium Grid infrastructure

For webship-js 1.0.x, Nightwatch.js was the right choice at the right time.

Where Nightwatch.js Hit the Wall

But then the web changed. And the problems started piling up.

Problem 1: The Selenium Bottleneck

Nightwatch.js talks to browsers through Selenium WebDriver. That means every command goes through an HTTP protocol layer:

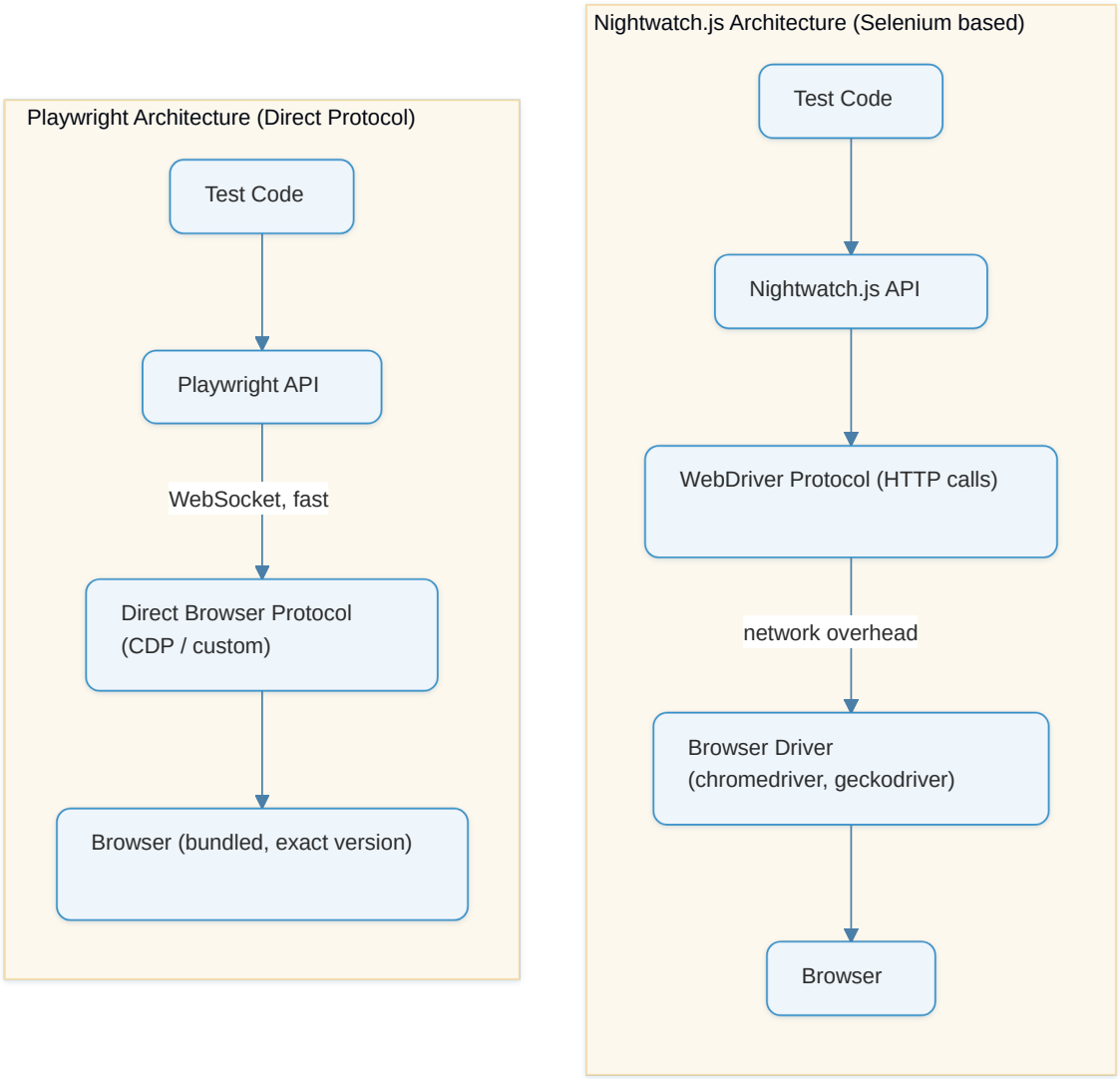


FIGURE A.2 *Problem 1: The Selenium Bottleneck*

Every single command in the Nightwatch.js stack is an HTTP request/response. Click a button? HTTP request. Read text? HTTP request. Check visibility? HTTP request. That adds up.

The Playwright stack keeps a persistent connection, so there is no HTTP overhead per command. Playwright controls the browser like a puppet master controls a puppet, directly, with no middleman.

The difference in speed is not marginal. It is dramatic. A test suite that takes 8 minutes on Nightwatch.js can run in 3 minutes on Playwright. When you are running tests on every push in CI, that difference is the difference between developers waiting and developers shipping.

Problem 2: Flaky Waits

This is the one that hurt the most. Nightwatch.js uses `waitForElementVisible` with a timeout:

```
// Nightwatch.js: hope-based waiting
browser.waitForElementVisible('#result', 10000);
// What if it takes 10001ms? Failure.
// What if it takes 50ms? You still configured for 10000ms.
// What if the element appears but the text hasn't loaded yet? Race
condition.
```

Playwright's approach is fundamentally different:

```
// Playwright: auto-waiting, built into every action
await page.click('#submit');
// Playwright automatically waits for the element to be:
// 1. Attached to the DOM
// 2. Visible
// 3. Stable (not animating)
// 4. Enabled (not disabled)
// 5. Not obscured by another element
// All of this happens automatically. No waitFor calls needed.
```

This is not a convenience feature. This is the single biggest reason Playwright tests are more reliable than Nightwatch.js tests. Every action in Playwright has auto-waiting built in. You do not sprinkle `waitForElementVisible` throughout your code and pray that the timeout is long enough. Playwright waits for exactly the right conditions, automatically.

In webship-js 2.0, this translates directly to the `wait for AJAX to finish` step that uses Playwright's `networkidle` wait state, something that was never reliable in the Selenium world.

Problem 3: Multi-Browser Reality

Nightwatch.js can run tests in Chrome, Firefox, and Edge through Selenium. But Safari? Only through Selenium Grid with specific configuration. And even then, the WebDriver implementations for each browser have different quirks, different timing, different bugs.

Multi-Browser Support Comparison

	Nightwatch.js	Playwright
Chromium	Via chromedriver	Bundled, exact version
Firefox	Via geckodriver	Bundled, exact version
WebKit/Safari	Painful / limited	Bundled, exact version
Mobile Safari	Not practical	WebKit engine included
Mobile Chrome	Via Appium (complex)	Chromium emulation
Driver management	Manual / webdriver-manager setup	Automatic. One command: npx playwright install
Browser versions	Whatever is installed on the machine	Bundled. Pinned. Reproducible.
"Works on my machine"	Common problem. Different driver versions cause chaos.	Rare. Same browser binary in dev and CI.

FIGURE A.3 *Problem 3: Multi-Browser Reality*

Playwright bundles the browsers. You run `npx playwright install` and you get Chromium, Firefox, and WebKit, the exact versions that Playwright was tested against. No driver version mismatches. No “it works in Chrome 120 but breaks in Chrome 121.” No fighting with geckodriver versions.

For webship-js 2.0, this means `BROWSER=webkit npx cucumber-js` just works. No extra setup. No extra configuration. No extra headaches.

Problem 4: Modern Web Features

Here is where Nightwatch.js really fell behind. The modern web has features that Selenium was never designed to handle:

Modern Web Features

Feature	`Nightwatch.js`	Playwright
Network interception	Not built-in	`route() API`
Request mocking	Third-party libs	Built-in
File downloads	Workarounds	Built-in
File uploads	Basic support	Full support + drag
Multiple tabs	Difficult	`context.newPage()`
iframes	Switchable	`frameLocator()`
Shadow DOM	Limited	Built-in support
Web Workers	No access	`evaluate() access`
Geolocation mock	Not supported	`context.setGeoloc()`
Permissions	Not supported	`context.grantPerms()`
Timezone override	Not supported	Built-in
Color scheme	Not supported	Built-in
Console log capture	Not built-in	page.on('console')
Network monitoring	Not built-in	page.on('request')
Video recording	Not built-in	Built-in
Tracing	Not built-in	Built-in

FIGURE A.4 *Problem 4: Modern Web Features*

Every one of these features exists in Playwright as a first-class API. No plugins. No workarounds. No “install this additional package and hope it works with your browser version.”

Problem 5: Debugging Experience

When a Nightwatch.js test fails, you get an error message and maybe a screenshot if you configured it. When a Playwright test fails, you can:

- Watch the test in headed mode with `HEADLESS=false`
- Slow it down with `SLOW_MO=500`
- Record a video of the entire test run
- Generate a trace file that lets you step through every action, see the DOM at each point, inspect network requests, and view console logs
- Use `page.pause()` to freeze execution and inspect the browser manually

The debugging experience alone would justify the switch.

The Migration: webship-js 1.0 to 2.0

When I decided to move webship-js from Nightwatch.js to Playwright, the goal was clear: keep the same Gherkin steps that teams already knew, but make them faster, more reliable, and more capable underneath.

The webship-js Migration

What STAYED the same:

- `Gherkin syntax (Given/When/Then)`
- Step text ("I fill in", "I press", "I should see")
- Cucumber-js as the BDD runner
- Feature file structure
- The testing philosophy (behavior over implementation)

What CHANGED:

- Browser engine: Selenium ▶ Playwright
- World class: NightwatchWorld ▶ PlaywrightWorld
- `Selectors: CSS/XPath via WebDriver` ▶ Playwright locators
- Waits: explicit timeouts ▶ auto-waiting
- Speed: 2-3x faster execution
- Reliability: dramatically fewer flaky tests

What teams experienced:

"We changed zero feature files. The Gherkin was identical.
But our CI pipeline went from 12 minutes to 4 minutes,
and our flaky test rate dropped from 8% to under 1%."

FIGURE A.5 *The Migration: webship-js 1.0 to 2.0*

That last point is critical. Because webship-js abstracts the browser automation behind Gherkin steps, the migration was invisible to anyone who only wrote feature files. The step definitions changed. The plumbing changed. But the language of the tests, the business language, stayed exactly the same.

This is the power of BDD done right. Your tests are not coupled to a tool. They are coupled to behavior. When the tool changes, the tests survive.

Side-by-Side: The Same Test, Two Engines

Let me show you the same test scenario, implemented with both engines:

The Feature File (Identical for Both)

```
Feature: Product Search
  Scenario: Search returns matching products
    Given I am on homepage
    When I fill in "Search" with "laptop"
    And wait for AJAX to finish
    Then I should see "Laptop Pro"
    And I should see "$999.99"
    And I should not see "Wireless Mouse"
```

Under the Hood: Nightwatch.js (webship-js 1.0)

```
// How the steps executed in webship-js 1.0 (Nightwatch.js)

// "I am on homepage"
browser.url(launchUrl);
browser.waitForElementVisible('body', 10000);

// "I fill in 'Search' with 'laptop'"
browser.waitForElementVisible('[name="Search"], #Search, .Search', 5000);
browser.setValue('[name="Search"]', 'laptop');

// "wait for AJAX to finish"
browser.pause(3000);  // <-- Yes, a hardcoded pause. Ugly but necessary.
// Nightwatch.js had no reliable "wait for all XHR to complete" mechanism.

// "I should see 'Laptop Pro'"
browser.assert.containsText('body', 'Laptop Pro');

// "I should not see 'Wireless Mouse'"
browser.expect.element('body').text.to.not.contain('Wireless Mouse');
```

Under the Hood: Playwright (webship-js 2.0)

```
// How the steps execute in webship-js 2.0 (Playwright)

// "I am on homepage"
await page.goto(launchUrl, { waitUntil: 'networkidle' });

// "I fill in 'Search' with 'laptop'"
const field = page.locator(buildSelector('Search'));
await field.fill('laptop');
// Playwright auto-waits for the field to be visible, enabled, and
editable

// "wait for AJAX to finish"
await page.waitForLoadState('networkidle');
// Waits until there are no network requests for 500ms. Not a guess. A
fact.

// "I should see 'Laptop Pro'"
await expect(page.locator('body')).toContainText('Laptop Pro');
// Playwright auto-retries this assertion until it passes or times out

// "I should not see 'Wireless Mouse'"
await expect(page.locator('body')).not.toContainText('Wireless Mouse');
```

See the difference? The Nightwatch.js version has a `pause(3000)`, a prayer that 3 seconds is enough for the AJAX call to complete. The Playwright version has `waitForLoadState('networkidle')`, a deterministic wait that resolves when the network is actually idle.

That `pause(3000)` is the root cause of more flaky tests than any other single line of code in the history of web testing. Sometimes the API responds in 200ms. Sometimes it takes 4 seconds because the database is under load. The pause does not care. It waits 3 seconds regardless, and when it is not enough, the test fails, not because the feature is broken, but because the wait was wrong.

Playwright eliminates this entire class of bugs.

What About Cypress?

People always ask me: “Why not Cypress?”

Cypress is an excellent tool. It has a beautiful developer experience, a time-travel debugger, and it made frontend testing fun for a generation of developers. I have respect for what the Cypress team built.

But Cypress has fundamental architectural limitations that make it wrong for webship-js:

Cypress vs Playwright for BDD Testing

Limitation	Cypress	Playwright
Multi-browser	Chrome-focused	Chromium+FF+WK
Multiple tabs	Not supported	Full support
Multiple domains	Workarounds	No restrictions
iframe support	Limited	`frameLocator()`
File download	Workarounds	Built-in
Hover actions	cy.trigger only	Native hover
Network layer	In-browser proxy	True interception
Execution model	In-browser JS	Out-of-process
Language	JavaScript only	JS, TS, Python, `Java, C#`
`BDD/Cucumber integration`	Third-party plugin	First-class with Cucumber-js

The deal-breaker for webship-js:
Cypress runs INSIDE the browser. This means it cannot `control things outside the browser tab, new tabs, downloads,` native dialogs, or cross-origin navigation.

Playwright runs OUTSIDE the browser. It controls the browser `like a user would, including everything the browser can do.`

FIGURE A.6 *What About Cypress?*

For unit-style component testing in a React or Vue project, Cypress is fantastic. For full end-to-end functional acceptance testing of a real application with logins, file uploads, multiple roles, modals, and AJAX calls, Playwright is the right tool.

The Numbers

Here are real-world numbers from a mid-sized web application (Drupal CMS with 45 feature files, 180 scenarios):

Before and After: Nightwatch.js ▶ Playwright

Metric	Nightwatch.js	Playwright
Full suite runtime	14 min 22 sec	5 min 08 sec
Flaky test rate	~6-8%	< 1%
Browser setup time	45 sec	12 sec
Failed due to timeouts	~15/month	~1/month
CI cost (compute time)	$340/month	$120/month
Browsers tested	Chrome, Firefox	Chromium, FF, WebKit
Debugging experience	Screenshots	Traces + Video
Feature files changed	0 out of 45	
Step definitions changed	All (rewritten for Playwright)	
Time to migrate	3 days	

ROI: Migration paid for itself in 2 months of saved CI costs.

FIGURE A.7 *The Numbers*

The 3x speed improvement is not magic. It comes from:

1. No HTTP overhead per command (direct protocol)
2. No hardcoded pauses (auto-waiting)
3. Faster browser launch (bundled, optimized)
4. Better parallelization (isolated browser contexts)

If You Are Migrating from Nightwatch.js

Here is the practical advice:

Step 1: Do Not Rewrite Your Feature Files

Your Gherkin is your specification. It does not change. If you are using webship-js, the feature files are identical between 1.0 and 2.0.

Step 2: Update Your Dependencies

```
# Remove Nightwatch.js dependencies
npm uninstall nightwatch chromedriver geckodriver selenium-server

# Install Playwright
npm install playwright
npx playwright install

# Update webship-js
npm install webship-js@latest
```

Step 3: Update Custom Step Definitions

If you wrote custom steps for Nightwatch.js, you need to rewrite them for Playwright:

```
// Nightwatch.js custom step (OLD)
When('I hover over the menu', function () {
  this.browser.moveToElement('#main-menu', 10, 10);
});

// Playwright custom step (NEW)
When('I hover over the menu', async function () {
  await this.page.hover('#main-menu');
});
```

The pattern is consistent:

- `this.browser` becomes `this.page`
- Callbacks become async/await
- Explicit waits become unnecessary (Playwright auto-waits)
- `setValue` becomes `fill`
- `click` becomes `click` (but with auto-waiting)

Step 4: Run Your Suite and Fix What Breaks

Most tests will pass immediately. The ones that fail are usually:

- Tests that relied on specific timing (remove hardcoded waits)
- Tests that used Nightwatch-specific APIs in custom steps
- Tests that assumed Selenium-specific behavior

Fix these one by one. It is a cleanup, not a rewrite.

The Verdict

I am not going to pretend this is a balanced comparison. It is not. Playwright is better than Nightwatch.js for automated functional acceptance testing in almost every dimension that matters:

- **Faster** (direct protocol, no Selenium overhead)
- **More reliable** (auto-waiting eliminates flaky tests)
- **Better multi-browser** (Chromium + Firefox + WebKit, bundled)
- **More capable** (network interception, video, tracing, geolocation, permissions)
- **Better debugging** (traces, headed mode, video recording)
- **Better maintained** (Microsoft backs it, monthly releases, growing ecosystem)
- **Future-proof** (the industry is moving toward Playwright)

Nightwatch.js served us well. It was the bridge that got us from “no automated testing” to “tests run on every push.” But the bridge has been crossed. Playwright is the destination.

If you are still on Nightwatch.js, you are not doing anything wrong. Your tests still run. Your bugs still get caught. But you are leaving speed, reliability, and capability on the table. The migration is measured in days, not months. And once you make it, you will wonder why you waited.

That is the story. webship-js 1.0 walked so webship-js 2.0 could run.

Appendix B: webship-js Step Definition Quick Reference

This is the cheat sheet. Print it. Pin it to your wall. Open it in a second monitor. These are all the built-in steps that webship-js gives you out of the box, organized by what you are trying to do.

Navigation

STEP	WHAT IT DOES
`Given I am on homepage`	Navigate to the base URL
`Given I am on "{path}"`	Navigate to a specific path
`When I go to "{path}"`	Navigate to a path (relative or absolute)
`When I go to the homepage`	Navigate to the base URL
`When I move backward one page`	Browser back button
`When I move forward one page`	Browser forward button
`When I reload the page`	Refresh the current page

Waiting

STEP	WHAT IT DOES
`And wait for the page to load`	Wait for `document.readyState` and `networkidle`
`And wait for AJAX to finish`	Wait for all XHR/Fetch requests to complete
`And wait {int}s`	Wait a specific number of seconds
`And wait for the modal to appear`	Wait for a visible modal/dialog
`And wait for the modal to close`	Wait for all modals to be hidden

Clicking

STEP	WHAT IT DOES
`When I click "{text}"`	Click a link or element by visible text
`When I press "{text}"`	Click a button by visible text or value
`When I click "{text}" in the "{row}" row`	Click a link in a specific table row
`When I press "{text}" in the "{row}" row`	Click a button in a specific table row
`When I click "{text}" in the modal`	Click a link inside the currently visible modal
`When I click "{text}" in the modal content`	Click inside the modal body
`When I press "{text}" in the modal`	Click a button inside the modal

Forms

STEP	WHAT IT DOES
`When I fill in "{field}" with "{value}"`	Type text into an input field
`When I select "{value}" from "{field}"`	Select an option from a dropdown
`When I additionally select "{value}" from "{field}"`	Select additional option (multi-select)
`When I check "{field}"`	Check a checkbox
`When I uncheck "{field}"`	Uncheck a checkbox
`When I attach the file "{path}" to "{field}"`	Upload a file to a file input

Assertions, Visibility

STEP	WHAT IT DOES
`Then I should see "{text}"`	Assert text is visible on the page
`Then I should not see "{text}"`	Assert text is NOT visible on the page
`Then I should see "{text}" in the "{region}" region`	Assert text in a specific region
`Then I should not see "{text}" in the "{region}" region`	Assert text NOT in a region
`Then I should see "{text}" in the modal`	Assert text inside the visible modal

Assertions, Elements

STEP	WHAT IT DOES
`Then the "{field}" field should contain "{value}"`	Assert input field has a value
`Then the "{field}" field should not contain "{value}"`	Assert input field does NOT have a value
`Then the checkbox "{field}" should be checked`	Assert checkbox is checked
`Then the checkbox "{field}" should not be checked`	Assert checkbox is unchecked
`Then the "{field}" select should have "{option}" selected`	Assert selected dropdown value

Assertions, URL and Title

STEP	WHAT IT DOES
`Then I should be on "{path}"`	Assert current URL matches
`Then the page title should contain "{text}"`	Assert page title contains text

Scrolling

STEP	WHAT IT DOES
`When I scroll to top`	Scroll to the top of the page
`When I scroll to bottom`	Scroll to the bottom of the page
`When I scroll to "{selector}"`	Scroll to a specific element

Modal and Dialog

STEP	WHAT IT DOES
`Then I should see a modal`	Assert a modal/dialog is visible
`Then I should not see a modal`	Assert no modal is visible
`When I close the modal`	Click the close button in the modal
`When I fill in "{field}" with "{value}" in the modal`	Fill a field inside the modal

Screenshots

STEP	WHAT IT DOES
`Then I take a screenshot`	Capture full-page screenshot
`Then I take a screenshot of "{selector}"`	Capture screenshot of specific element

API Testing (webship-api.js)

STEP	WHAT IT DOES
`Given the API base URL is "{url}"`	Set the base URL for API requests
`Given I am authenticating as "{user}" with "{pass}" password`	Set Basic Auth
`When I send a GET request to "{path}"`	Send GET request
`When I send a POST request to "{path}" with body:`	Send POST with JSON body
`When I send a PUT request to "{path}" with body:`	Send PUT with JSON body
`When I send a PATCH request to "{path}" with body:`	Send PATCH with JSON body
`When I send a DELETE request to "{path}"`	Send DELETE request
`Then the API response code should be {int}`	Assert HTTP status code
`Then the response should be valid JSON`	Assert response is parseable JSON
`Then the API response should contain "{text}"`	Assert response body contains text
`Then the JSON response should have "{key}" equal to "{value}"`	Assert JSON property value
`Then the JSON response should have property "{key}"`	Assert JSON property exists
`Then the JSON response should not have property "{key}"`	Assert JSON property absent

Environment Variables

VARIABLE	DEFAULT	WHAT IT DOES
`BROWSER`	`chromium`	Browser engine: `chromium`, `firefox`, `webkit`
`HEADLESS`	`true`	Run headed: `HEADLESS=false`
`SLOW_MO`	`300`	Milliseconds delay between actions
`LAUNCH_URL`	from config	Override the base URL

Common Patterns

Login and Do Something

```
Given I am a logged in user with the "admin" user
When I go to "/admin/products"
And wait for AJAX to finish
Then I should see "Products"
```

Fill a Form and Submit

```
When I fill in "Name" with "Jane Doe"
And I fill in "Email" with "jane@example.com"
And I select "Premium" from "Plan"
And I check "I agree to the terms"
And I press "Submit"
And wait for AJAX to finish
Then I should see "Registration complete"
```

Test a Table

```
Then I should see "Laptop Pro"
When I click "Edit" in the "Laptop Pro" row
And wait for AJAX to finish
And I fill in "Price" with "899.99"
And I press "Save"
And wait for AJAX to finish
Then I should see "Product updated"
```

Test a Modal

```
When I press "Delete"
And wait for the modal to appear
Then I should see "Are you sure?" in the modal
When I press "Confirm" in the modal
And wait for AJAX to finish
Then I should see "Deleted"
```

API Call

```
Given the API base URL is "http://localhost:8080"
When I send a GET request to "/api/products"
Then the API response code should be 200
And the JSON response should have property "data"
```

Web-First Assertions (with `within N seconds`)

STEP	WHAT IT DOES
`Then "sel" should be visible within {N} seconds`	Poll until visible
`Then "sel" should be hidden within {N} seconds`	Poll until hidden
`Then "sel" should be attached`	Element present in DOM
`Then "sel" should be focused`	Element has keyboard focus
`Then "sel" should be enabled` / `disabled`	Interactive state
`Then "sel" should be editable`	Editable (not readonly/disabled)
`Then "sel" should contain text "X"`	Substring match
`Then "sel" should have text "X"`	Exact text match
`Then "sel" should have value "X"`	Input value
`Then "sel" should have class "X"`	Class present
`Then "sel" should have attribute "X" with value "Y"`	Attribute match
`Then "sel" should have a count of {N}`	Match count
`Then "sel" should be in the viewport`	Element in visible viewport
`Then the "X" button should be visible`	Role-aware

Default budget 5s. Override with trailing `within N seconds`.

Iframes

STEP	WHAT IT DOES
`When I switch to the iframe "name"`	Enter iframe by `name` attr
`When I switch to the iframe with title "X"`	Enter iframe by `title`
`When I switch to iframe with locator "sel"`	Enter iframe by CSS
`When I switch to the root document`	Return to top-level page
`When I click "X" inside the iframe`	Click inside current iframe
`When I fill in "X" with "Y" inside the iframe`	Fill inside iframe
`Then I should see "X" inside the iframe`	Text assertion inside iframe

Clock and Time Mocking

STEP	WHAT IT DOES
`Given the system time is "X"`	Freeze in-page clock at instant
`When I advance the clock by {N} ms`	Step clock forward (also: seconds, minutes)
`When I pause the clock`	Stop clock progression
`When I resume the clock`	Resume real-time
`When I wait until pending timers settle`	Fast-forward all queued timers

Network Mocking

STEP	WHAT IT DOES
`Given the URL "X" is blocked`	Block request to URL
`Given the URL "X" is delayed by {N} ms`	Delay response
`Given the URL "X" returns status {N}`	Mock status code
`Given the URL "X" returns status {N} with body "Y"`	Mock status + body
`Given the URL "X" returns the JSON:`	Mock JSON response
`Given the network is offline` / `online`	Toggle browser online state

Network Recording

STEP	WHAT IT DOES
`Given I start recording network requests`	Start request recorder
`Then a {METHOD} request to "X" should have been made`	Assert request fired
`Then a request to "X" should have been made`	Method-less assert
`Then no request to "X" should have been made`	Negation

File Downloads and Video

STEP	WHAT IT DOES
`When I download the file from the link "X"`	Trigger download from link
`When I download the file from the URL "X"`	Trigger download from URL
`Then the downloaded file name should be "X"`	Exact filename
`Then the downloaded file name should contain "X"`	Substring filename
`Then the downloaded file should contain:`	Substring content
`Then the downloaded file should be a zip archive containing the following files named:`	ZIP entry check
`Then the downloaded file should be a zip archive not containing the following files partially named:`	Negative ZIP check
`Given I start video recording`	Begin video capture
`When I stop video recording`	Stop video capture
`When I save the current video as "X"`	Save with custom name

XML Response

STEP	WHAT IT DOES
`Then the response should be in XML format`	Detect XML response
`Then the XML element "xpath" should exist`	Element exists
`Then the XML element "xpath" should be equal to "X"`	Element value
`Then the XML element "xpath" should contain "X"`	Element substring
`Then the XML element "xpath" should have {N} element(s)`	Element count
`Then the XML attribute "X" on element "xpath" should be equal to "Y"`	Attribute value
`Then the XML should use the namespace "X"`	Namespace check

YAML Response

STEP	WHAT IT DOES
`Then the response should be in YAML format`	Detect YAML response
`Then the YAML element "path" should exist`	Element exists
`Then the YAML element "path" should be equal to "X"`	Value match
`Then the YAML value at "path" should be of type "X"`	Type check
`Then the YAML value at "path" should be greater than {N}`	Numeric range
`Then the YAML array at "path" should contain an item where "K" is "V"`	Array predicate
`Then the YAML keys at "path" should be exactly "K1,K2,K3"`	Strict keys
`Then the YAML response should have {N} document(s)`	Multi-document count
`Then the YAML should match JSON Schema "path"`	Schema validate
`Then the YAML should have no duplicate keys`	Lint check

Pointer Actions and Keyboard

STEP	WHAT IT DOES
`When I double-click on "X"`	Native dblclick
`When I right-click on "X"`	Context menu
`When I middle-click on "X"`	Middle button
`When I hover over "X"`	Mouse hover
`When I tap on "X"`	Mobile touch tap
`When I drag "A" to "B"`	Drag-and-drop
`When I click on "X" while holding "Shift"`	Click with modifier
`When I press the key "X"`	Single key press
`When I press the keys "Control+S"`	Key chord
`When I press the keys "X" on the element "sel"`	Scoped chord

Advanced Field Types

STEP	WHAT IT DOES
`When I fill in the color field "X" with the value "Y"`	Color picker
`When I fill in the datetime field "X" with date "D" and time "T"`	Datetime
`When I fill in the date part of the datetime field "X" with "D"`	Date only
`When I fill in the start datetime field "X" with date "D" and time "T"`	Range start
`When I fill in the end datetime field "X" with date "D" and time "T"`	Range end
`When I fill in the multi-value field "X" with the following values:`	Tag chips
`When I fill in the WYSIWYG field "X" with the "Y"`	Rich editor
`When I select "X" text in "Y" field`	Substring select
`When I select from {N} to {M} text in "Y" field`	Range select
`When I select all text in "Y" field`	Select all
`Then the field "X" should be required` / `empty` / `not empty`	Field state
`Then the field "X" should have "valid" state`	Validity state

JavaScript Console Health

STEP	WHAT IT DOES
`Then there should be no JavaScript errors`	Fail on any console error
`Then there should be no JavaScript warnings`	Fail on any console warning
`Then JavaScript errors should not match "pattern"`	Fail on matching error
`Then print JavaScript errors`	Dump errors to stdout

Modes: `fail` (default), `warn`, `off`, set via `this.javascriptErrorMode` in a Before hook.

Cookies (Four-Way Matrix)

NAME	VALUE	STEP (EXISTENCE)
exact	exact	`a cookie with the name "X" and the value "Y" should exist`
exact	contains	`a cookie with the name "X" and a value containing "Y" should exist`
contains	exact	`a cookie with a name containing "X" and the value "Y" should exist`
contains	contains	`a cookie with a name containing "X" and a value containing "Y" should exist`
(any)	(any)	`a cookie with the name "X" should exist` (name-only)

All have `should not exist` negations. Setters: `the cookie "X" is set to "Y"`, `the cookie "X" is removed`, `all cookies are cleared`.

Storage

STEP	WHAT IT DOES
`Given the local storage "X" is set to "Y"`	Seed localStorage key
`Given the local storage "X" is removed`	Remove single key
`Given local storage is cleared`	Clear all localStorage
`Given the session storage "X" is set to "Y"`	Seed sessionStorage key
`Given the session storage "X" is removed`	Remove single key
`Given session storage is cleared`	Clear all sessionStorage

Named Selectors

STEP	WHAT IT DOES
`Given I add "name" selector for "css" css selector`	Inline CSS selector
`Given I add "name" selector for "xpath" xpath selector`	Inline XPath selector
`Given I define css selectors:` (with table)	Bulk CSS
`Given I define xpath selectors:` (with table)	Bulk XPath
`Given I add selectors from "file.json" file`	Load from file
`Then print css selectors` / `print xpath selectors`	Dump registry

Spatial & Layout

STEP	WHAT IT DOES
`Then the element "X" should be displayed within a viewport`	In current viewport
`Then the element "X" should be at the top of the viewport`	Sticky/pinned
`Then the element "X" should be centered in the viewport`	Centred element
`Then the element "X" should appear after the element "Y"`	DOM ordering
`Then the text "X" should appear after the text "Y"`	Text ordering
`Then "X" should contain cssproperty "prop" with value "val"`	Computed style

CI/CD Providers Shipped

GitHub Actions · GitLab CI · Bitbucket Pipelines · CircleCI · Travis CI · Jenkins · Azure Pipelines · AWS CodeBuild · Google Cloud Build · TeamCity (Kotlin DSL) · Semaphore · Drone CI · Woodpecker CI · Forgejo Actions · Harness CI · Bamboo Data Center · Codefresh · Octopus Deploy (CD-only) · Codeship (retired)

See Chapter 45 for full per-provider configs.

Appendix D: Further Reading

A curated bibliography of books, articles, and research papers that shaped modern automated acceptance testing. Several of these were read directly during the writing of this book and quoted in Chapter 51. The rest are foundational works any serious test-suite practitioner should know.

The bibliography is grouped by theme. Within each group, items are ordered by influence and chronology, earlier and more-cited works appear first.

Foundational TDD

Kent Beck, *Test-Driven Development by Example*

Addison-Wesley, 2002. ISBN 978-0321146533.

The book that named TDD. Beck walks through two extended examples (a money library and the development of xUnit itself), introducing the red-green-refactor cycle, triangulation, the “obvious implementation” technique, and the discipline of small steps. Beck’s prose is conversational; the book reads as a transcript of Beck thinking aloud. If you read only one book on testing, read this one.

Key phrase: *clean code that works.*

Steve Freeman & Nat Pryce, *Growing Object-Oriented Software, Guided by Tests* (GOOS)

Addison-Wesley, 2009. ISBN 978-0321503626.

The TDD book that addresses whole-system design, not just unit-level rhythm. Introduces the **walking skeleton** concept, the **listen to the tests** principle, and the use of **mock objects** as a design tool. The first half is theory; the second half is a 100-page worked example.

Foreword by Kent Beck. Recommended reading after Beck's *TDD by Example*.

Jeff Langr, *Modern C++ Programming with Test-Driven Development*

Pragmatic Bookshelf, 2013. ISBN 978-1937785482.

A demonstration that TDD applies even to system-level languages with no GC and explicit memory management. Useful for senior engineers who think "TDD doesn't work in my language." Langr proves it does.

Harry Percival, *Test-Driven Development with Python*

O'Reilly, 2014 (1st ed.), 2017 (2nd ed.). ISBN 978-1491958704.

A walkthrough of building a Django web app using rigorous TDD throughout, from functional Selenium tests down to model-level units. Introduces the **"Obey the Testing Goat"** mantra. Excellent for web developers new to TDD.

The free online version at https://www.obeythetestinggoat.com is the same content as the print book.

Michael Feathers, *Working Effectively with Legacy Code*

Prentice Hall, 2004. ISBN 978-0131177055.

Feathers defines legacy code as "code without tests" and provides 24 techniques for getting untested code under test. Essential reading for anyone joining a codebase that wasn't built test-first. The dependency-breaking techniques (Extract Interface, Subclass and Override Method, Pass Null) are timeless.

Roy Osherove, *The Art of Unit Testing*

Manning, 2009 (1st ed.), 2013 (2nd ed.), 2024 (3rd ed., with Vladimir Khorikov). ISBN 978-1617297489.

A pragmatic guide to unit-test design: naming, structure, maintainability, the three pillars (Trustworthy, Maintainable, Readable). Covers fakes, stubs, and mocks in depth. The 3rd edition adds JavaScript/Jest examples.

BDD and Specification by Example

Dan North, “Introducing BDD” (2006) and “Behaviour-Driven Development” (2009)

Articles available at https://dannorth.net/introducing-bdd/

North’s original essays defining BDD. The 2006 piece reframes TDD around “behaviour” rather than “test,” introduces the Given/When/Then vocabulary, and seeds what would become the JBehave and RBehave frameworks (predecessors to Cucumber).

A 12-minute read that shaped a decade of testing tools.

Gojko Adzic, *Specification by Example*

Manning, 2011. ISBN 978-1617290084.

Adzic interviewed 50+ teams running BDD-style processes and distilled the patterns into seven practices: deriving scope from goals, illustrating with examples, refining specifications, automating without changing them, validating frequently, evolving a documentation system, and changing the process to encode learning.

The book that took BDD from a developer’s vocabulary to a whole-team workflow.

Gojko Adzic, *Bridging the Communication Gap*

Neuri, 2009. ISBN 978-0955683619.

Adzic's earlier, slimmer book on the same theme: how to use concrete examples to align engineering, product, and business. Less comprehensive than *Specification by Example*; faster to read.

Matt Wynne & Aslak Hellesøy, *The Cucumber Book*

Pragmatic Bookshelf, 2012 (1st ed.), 2017 (2nd ed., with Steve Tooke). ISBN 978-1680502381.

The canonical reference for Cucumber. Covers Gherkin basics, expressive scenarios, step definitions, test data, transformations, hooks, tagging, and a worked example. The second edition adds Cucumber-JVM and other ports.

Foreword by Dan North: "I hope you enjoy this book as much as I did."

Ian Dees, Matt Wynne & Aslak Hellesøy, *Cucumber Recipes*

Pragmatic Bookshelf, 2013. ISBN 978-1937785017.

A recipe-format companion to *The Cucumber Book*. Each entry follows Problem / Ingredients / Solution. Covers polyglot Cucumber (Ruby, Java, C#, JavaScript), table transformations, REST APIs, browser automation, and mobile. Heavily inspired the recipe format used in *this* book.

Foreword by Ward Cunningham.

David Chelimsky, Dave Astels, Zach Dennis, Aslak Hellesøy, Bryan Helmkamp & Dan North, *The RSpec Book*

Pragmatic Bookshelf, 2010. ISBN 978-1934356371.

Despite the title, this is not just an RSpec reference. The book covers BDD philosophy, then RSpec and Cucumber together, then how to apply BDD to Rails. Foreword by Robert C. Martin, the famous "The meme is Craftsmanship" essay.

Liz Keogh, BDD essays and talks (2008-present)

Articles at https://lizkeogh.com

Keogh has been writing and speaking on BDD since the early days. Her insight: “BDD is about the conversation, not about the tools.” She emphasises that the scenarios are *artefacts* of the conversation, the conversation itself is where the value is generated.

Key talks: “BDD: A Three-Headed Monster”, “Cynefin for Devs”, “Estimating in Complexity”.

Enrique Amodeo, *Learning Behavior-Driven Development with JavaScript*

Packt, 2015. ISBN 978-1784392642.

A practical JavaScript-focused BDD book. Covers Mocha, Chai, Sinon, Cucumber.js, REST API testing, WebDriverJS, Protractor, and the Page Object pattern. Useful for JavaScript developers new to BDD.

Notable for its pragmatic stance: “The real world calls for a more pragmatic approach: write the tests that really pay off and do not write those that do not give you much value.”

Wojciech Bulaty, *Instant Cucumber BDD How-to*

Packt, 2013. ISBN 978-1782165156.

A short (~70 page) practical introduction to Cucumber. Worth knowing if you want the simplest possible “what is Cucumber?” book. Less comprehensive than the Pragmatic Bookshelf titles.

Behat user guide, Konstantin Kudryashov & contributors

Available at https://docs.behat.org

Behat is the PHP port of Cucumber. The user guide is a reference for PHP teams; the principles transfer cleanly to webship-js and other Cucumber-family runners.

Jeff Foster, “Behaviour-Driven Development with Cucumber”

Article in The Java Specialists Newsletter, 2013.

A short article (~12 pages) that walks through a Java Cucumber example. Useful as a quick orientation for JVM teams; nothing more.

Agile Testing and Process

Lisa Crispin & Janet Gregory, *Agile Testing*

Addison-Wesley, 2009. ISBN 978-0321534460.

The book that brought “agile testing” into mainstream practice. Introduces the **Agile Testing Quadrants** (technology-facing vs business-facing, supporting development vs critiquing product), a vocabulary that’s now standard.

Lisa Crispin & Janet Gregory, *More Agile Testing*

Addison-Wesley, 2014. ISBN 978-0321967053.

Sequel to *Agile Testing*. Updates the quadrants for the post-DevOps era, covers test environments, mobile testing, and continuous delivery testing.

Mike Cohn, *Succeeding with Agile*

Addison-Wesley, 2009. ISBN 978-0321579362.

The book that introduced the **Test Pyramid**, many fast unit tests at the base, fewer slow end-to-end tests at the top. Chapter 17 (“Stating Goals”, “Building a Quality Backbone”) is the relevant section for testing. Cohn’s broader work on user stories and estimation also informs scenario-driven workflows.

Kent Beck, *Extreme Programming Explained: Embrace Change*

Addison-Wesley, 1999 (1st ed.), 2004 (2nd ed.). ISBN 978-0321278654.

The XP manifesto. Test-first development is one of XP's core practices, but the book covers the wider system: pair programming, continuous integration, on-site customer, planning game. Background reading for understanding *why* TDD is a discipline rather than a checklist.

Test Doubles and Mocking

Gerard Meszaros, *xUnit Test Patterns: Refactoring Test Code*

Addison-Wesley, 2007. ISBN 978-0131495050.

A 900-page pattern catalogue covering test smells, test doubles (the **dummy / fake / stub / spy / mock** taxonomy), and refactorings to clean up test code. Cited everywhere; rarely read cover to cover. Use as a reference.

Martin Fowler, "Mocks Aren't Stubs" (2007)

Article at https://martinfowler.com/articles/mocksArentStubs.html

A short essay that clarifies the mockist-vs-classicist TDD debate. Fowler defines stubs (return canned answers) and mocks (verify interactions) and argues each style produces different designs.

Martin Fowler, "TestPyramid" (2012)

Article at https://martinfowler.com/bliki/TestPyramid.html

Fowler's short summary of Mike Cohn's pyramid. Often cited; one page.

Martin Fowler, “Refactoring” (book, 1999 and 2018)

Addison-Wesley. ISBN 978-0134757599 (2nd ed.).

The book on refactoring. Relevant to testing because every refactor needs a safety net of tests. The 2nd edition uses JavaScript; the 1st used Java.

Domain-Driven Design and Ubiquitous Language

Eric Evans, *Domain-Driven Design*

Addison-Wesley, 2003. ISBN 978-0321125217.

The book that introduced “ubiquitous language”, the principle that the team uses the same terminology in conversation, documentation, and code. Dan North cites DDD as a direct influence on BDD vocabulary.

A dense 500-page book; the first 100 pages are the most essential.

Vaughn Vernon, *Implementing Domain-Driven Design*

Addison-Wesley, 2013. ISBN 978-0321834577.

The more practical companion to Evans. Easier to read; same vocabulary.

Property-Based and Mutation Testing

Fred Hebert, *Property-Based Testing with PropEr, Erlang, and Elixir*

Pragmatic Bookshelf, 2019. ISBN 978-1680506211.

The most accessible introduction to property-based testing for any language. Hebert teaches how to think in properties rather than examples, a skill that complements (not replaces) example-based BDD.

John Hughes, "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs" (2000)

Research paper, ICFP 2000.

The original property-based testing paper. Hughes's QuickCheck for Haskell inspired Hypothesis (Python), fast-check (JavaScript), PropEr (Erlang), and many others.

Roy Osherove, discussion of mutation testing in *The Art of Unit Testing*

See above. Osherove's treatment of mutation testing is a practical entry point.

Tim Pope, Marick et al., mutation testing tools

Stryker (JavaScript): https://stryker-mutator.io PIT (Java): https://pitest.org Mutmut (Python): https://mutmut.readthedocs.io

The most-cited mutation testing tools. Documentation reads quickly; the technique is straightforward to add to a CI pipeline.

Continuous Delivery and DevOps

Jez Humble & David Farley, *Continuous Delivery*

Addison-Wesley, 2010. ISBN 978-0321601919.

The book that established CD practices. Acceptance test automation is one of CD's core building blocks; this book describes how it fits into pipelines, environments, and deployment.

Gene Kim, Patrick Debois, John Willis & Jez Humble, *The DevOps Handbook*

IT Revolution, 2016. ISBN 978-1942788003.

The broader DevOps reference. Testing is covered as part of the deployment pipeline, with emphasis on shifting tests left (run them earlier) and parallel execution.

Research Papers Worth Reading

David Saff & Michael D. Ernst, "An Experimental Evaluation of Continuous Testing during Development" (2004)

ISSTA 2004 proceedings.

Saff (later of "Saff Squeeze" fame) ran experiments showing that running tests continuously during editing reduces development time. The paper informs modern test-on-save workflows.

Steve Freeman, Tim Mackinnon, Nat Pryce & Joe Walnes, "Mock Roles, not Objects" (2004)

OOPSLA 2004.

The paper that introduced the “mock roles” perspective: mock the **role** a dependency plays, not the object itself. Predecessor to *Growing Object-Oriented Software*.

Lasse Koskela, *Effective Unit Testing*

Manning, 2013. ISBN 978-1935182573.

A pattern catalogue for unit-test quality, with examples in Java. Covers test smells, idioms, and the FIRST principles (Fast, Independent, Repeatable, Self-validating, Timely).

Kent C. Dodds, “The Testing Trophy and Testing Classifications” (2018)

Article at https://kentcdodds.com/blog/the-testing-trophy-and-testing-classifications

The modern revisionist take on the test pyramid. Dodds argues for more integration tests and fewer mocked unit tests. Influential in JavaScript testing communities.

Andre Schaffer, “Testing of Microservices” (2018)

Spotify Engineering blog.

Introduces the **testing honeycomb**, an alternative shape for microservices where integration tests dominate. Useful counterpoint to the pyramid.

Standards and Specifications

Cucumber.io, Gherkin Reference

https://cucumber.io/docs/gherkin/reference/

The authoritative Gherkin syntax reference, maintained by the Cucumber maintainers. Treat as the spec when in doubt about a syntactic question.

W3C, WCAG 2.2

https://www.w3.org/TR/WCAG22/

The accessibility standard webship-js's a11y checks validate against. Levels A, AA, AAA. Critical for any user-facing test suite.

W3C, WebDriver BiDi

https://www.w3.org/TR/webdriver-bidi/

The bidirectional WebDriver protocol that underpins modern browser automation (Playwright, Puppeteer, Selenium 4). Relevant when debugging cross-tool compatibility issues.

Official Tool Documentation

This book stays focused on the recipes, the behaviour you want to verify and the Gherkin that verifies it, rather than re-documenting the tools underneath. When you need the underlying API, the canonical sources are below. Bookmark them; they are the authoritative reference the recipes build on.

Playwright

https://playwright.dev

The browser-automation engine webship-js drives. Reach for it when you need the low-level locator, network, clock, or tracing API behind a step, the auto-waiting model, `page.route()`, `page.clock`, and the trace viewer are all documented here.

Cucumber-js

https://github.com/cucumber/cucumber-js · guide at **https://cucumber.io/docs/cucumber/**

The BDD runner that executes the feature files. The docs cover Gherkin syntax, tag expressions, hooks, the World object, and the `--parallel` runner, the mechanics the architecture chapters lean on.

axe-core

https://github.com/dequelabs/axe-core

The accessibility engine behind the a11y steps (Chapter 15). The rule reference explains what each WCAG check actually tests.

Node.js

https://nodejs.org/en/docs

The runtime. Useful mostly for the version policy and the built-in modules custom steps occasionally use.

Webship.co Resources

Webship.co Documentation

https://webship.co/docs/webship-js/2.0.x/

The canonical webship-js reference. Includes the BBR smart-waits guide, the step-definition reference, the API testing guide, the accessibility guide, the CI/CD guide for 19 providers, and the AI-agent guide.

Webship.co Blog

https://webship.co/blog

Articles by Rajab Natshah and the Webship team on automated functional acceptance testing, AI-powered tooling, and Drupal CMS testing. Most articles include working webship-js examples.

Webship-js GitHub Organisation

https://github.com/webship

The webship-js source, the AI agent, the Claude plugin, the skill set, the example automated-testing repo, and the documentation. Read the example repos for production-grade Gherkin patterns.

Vardot/varbase Test Suite

https://github.com/Vardot/varbase/tree/10.1.x/tests/features/varbase

The Varbase acceptance suite (covered in Chapter 47). A fresh Varbase 11 install ships a fifteen-group test matrix, from base requirements through content, workflow, AI authoring, and a dedicated quality group. One of the most-tested Drupal distributions in the wild; reading the feature files is a free seminar in Gherkin patterns.

Vardot/varbase-project Test Suite

https://github.com/Vardot/varbase-project/tree/11.0.x/tests/features

The Drupal 11 evolution of the same suite. Demonstrates upgrade-path testing patterns and modernised conventions.

webshipco/webshipco-automated-testing

https://github.com/webshipco/webshipco-automated-testing

The small webshipco.co marketing-site test suite used as the case study in Chapter 50. Two custom steps, ten feature files. Reading the repo in 20 minutes is the cheapest possible TDD case study.

Reading Plan

A practical reading order if you're new to the field:

1. **Beck, *TDD by Example***, the foundational rhythm
2. **Dan North, "Introducing BDD"** (essay, 12 minutes), the BDD reframe
3. **Wynne & Hellesøy, *The Cucumber Book***, the operational reference
4. **Adzic, *Specification by Example***, the team-level workflow
5. **Freeman & Pryce, *GOOS***, the design-level view
6. **Crispin & Gregory, *Agile Testing***, the QA partnership view
7. **Feathers, *Working Effectively with Legacy Code***, the rescue manual

Add Fowler's *Refactoring* and Evans's *Domain-Driven Design* whenever you find the time. Both are deep and unrushed.

If you read all seven, you'll have spent roughly 50 hours and inherited the practical wisdom of three generations of practitioners. The compounding effect on your engineering career is hard to overstate.

A Final Note

The works listed above are the ones that have most directly shaped webship-js's design and this book's voice. The list is opinionated and incomplete. Every reader will find their own canon over time. Build it deliberately. The bibliography you can quote from memory is the toolkit you actually use.

About the Authors

This book was written by three of the five Natshah siblings, Rajab, Tasneem, and Shaimaa, who build open-source web-testing tools together at **Webship.co**.

Rajab Natshah

Lead author of this book and the original author of webship-js, which he created in 2021. A Drupal contributor since 2011 and a senior engineer / Technical Product Lead at **Vardot**, he was ranked among the top Drupal contributors worldwide for his work readying **Varbase**, the enterprise distribution he maintains, for Drupal 9. His public profile spans more than a hundred maintained modules and tools, from Behat UI and Cucumber UI to Bootstrap Layout Builder, and now webship-js on Playwright and Cucumber-js. He is a founder of **Webship.co** and a long-time community mentor and conference speaker.

- GitHub: github.com/RajabNatshah · Drupal.org: drupal.org/u/rajab-natshah · Webship.co: webship.co

Tasneem Natshah

Co-founder of Webship.co and a long-time Drupal contributor, with credited work on the Cucumber and WebEditor projects among others. She builds the Drupal-integrated layers that connect real sites to webship-js test suites, the Cucumber UI module and the test-runner integration are her territory, and the modules that quietly make the rest work.

- GitHub: github.com/TasneemNatshah · Drupal.org: drupal.org/u/tasneem-natshah

Shaimaa Natshah

Co-founder and CEO of Webship.co and a contributor across the Webship and Cucumber projects. She drives the product side, the website, the documentation portal, the visual identity, and the writing that turns the code into something a team can adopt without a PhD in test theory. The fact that the webship.co docs read like a book and not an API dump is largely her doing.

- GitHub: github.com/ShaimaaNatshah · Drupal.org: drupal.org/u/shaimaa-natshah

The people we owe beyond this team, our father, Mohammed Razem, and the wider open-source community, are thanked in the Dedication and Acknowledgments at the front of this book.

Articles & tutorials: webship.co/blog

AUTOMATED FUNCTIONAL TESTING RECIPES

Test the user journey, not the framework.

Plain-English Gherkin on top of Playwright and Cucumber-js. Write the step, run the suite, ship with confidence -- 700+ recipes covering navigation, forms, APIs, accessibility, CI/CD, and the AI-powered testing ecosystem.

© Rajab · Tasneem · Shaimaa Natshah