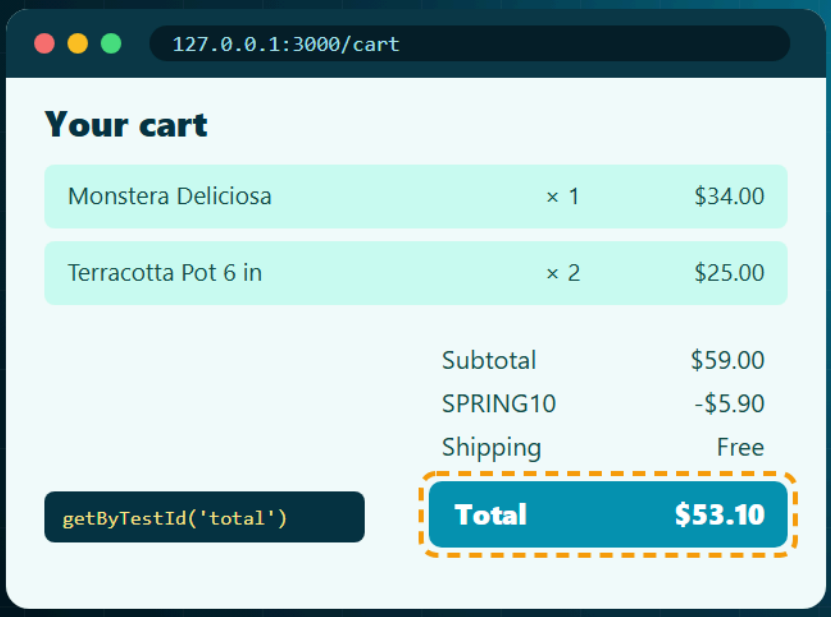
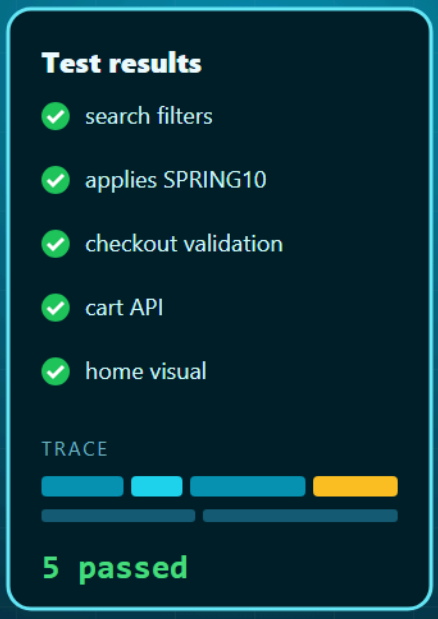





```
test('applies SPRING10', async ({ page }) => {
  await page.getByLabel('Coupon code').fill('SPRING10');
  await page.getByRole('button', { name: 'Apply' }).click();
  await expect(page.getByTestId('total')).toHaveText('$53.10');
});
```

FROM FIRST SCRIPT TO CI

Playwright End-to-End Testing

Reliable UI, API, and visual tests from your first script to CI

Locators · Assertions · Page Objects · Fixtures · API Tests
Mocking · Visual Regression · Traces · Sharding · CI Pipelines

TYPESCRIPT · CHROMIUM · FIREFOX · WEBKIT

THENEXBOOKS

Playwright End-to-End Testing

Reliable UI, API, and Visual Tests from Your First Script to CI
Locators, assertions, page objects, fixtures, mocking, visual regression, debugging, and pipelines — built step by step

First Edition

THENEXBOOKS

Playwright End-to-End Testing
Reliable UI, API, and Visual Tests from Your First Script to CI

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples and the companion Fernway Shop app may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Playwright is under active development and releases new versions frequently; command-line options, configuration settings, and reports can change between versions. Always consult the official documentation and release notes for the version you use.

Playwright is an open-source project maintained by Microsoft. Microsoft, Windows, Azure, Visual Studio Code, and Microsoft Edge are trademarks of the Microsoft group of companies. Chrome and Chromium are trademarks of Google LLC. Firefox is a trademark of the Mozilla Foundation. Safari and WebKit are trademarks of Apple Inc. GitHub and GitHub Actions are trademarks of GitHub, Inc. Node.js is a trademark of the OpenJS Foundation. Docker is a trademark of Docker, Inc. Other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by Microsoft or any other company mentioned.

Fernway Shop, the online plant store used as the running example in this book, is fictional, as are its products, customers, orders, and all sample data. Email addresses use the reserved example.com domain. The Fernway Shop app is a learning tool; do not deploy it to the internet.

Preface

Every team that ships a web application eventually asks the same question: does it still work? A developer changes a stylesheet and the checkout button disappears on narrow screens. A refactoring renames an API field and the cart shows `$NaN`. A new login provider works for everyone except the admin account. Unit tests are fast and valuable, but they cannot tell you whether a real person, in a real browser, can still sign in, find a product, and pay for it.

End-to-end tests answer that question by driving the application the way a user does. For years they also had a reputation for being slow, brittle, and hard to debug — tests that failed one run in ten for no visible reason, sleeps sprinkled everywhere, and failures nobody could reproduce. Playwright was designed to remove those problems at the root. It waits automatically for elements to be ready, retries assertions until the page reaches the expected state, isolates every test in a fresh browser context, runs tests in parallel by default, and records a complete trace of every action so a failure on a CI server can be replayed step by step on your own machine.

This book teaches Playwright Test from the very first command to a professional test suite running in continuous integration. We do not just list APIs. We test one realistic application from start to finish, introduce each feature at the moment it solves a problem we actually have, and explain *why* it works the way it does, so you can design reliable tests for your own applications.

Who This Book Is For

- **Developers** who want confidence that their features work end to end, and want tests that run fast and fail for real reasons.
- **QA engineers and testers** moving from manual testing or from older automation tools to a modern, code-based framework.
- **Team leads and DevOps engineers** who need to run browser tests reliably in CI pipelines and make sense of the results.
- **Students and career changers** who know basic JavaScript or TypeScript and want a practical, in-demand testing skill.

You should be comfortable with basic JavaScript (variables, functions, objects, `async` / `await`) and know what HTML elements and HTTP requests are. You do not need prior experience with Playwright, TypeScript's advanced features, or any other test framework.

What You Will Learn

- How to install Playwright, run your first test, and read every part of the output.
- How to find elements the way users do with role, label, and text locators — and why that makes tests resilient.

- How auto-waiting and web-first assertions remove timing problems, and how to test forms, dialogs, new tabs, frames, uploads, and downloads.
- How to structure a growing suite with hooks, tags, authentication state, page objects, custom fixtures, and well-managed test data.
- How to test REST APIs directly, combine API and UI steps for speed, intercept and mock network traffic, and catch visual regressions with screenshots.
- How to check accessibility, control time, and emulate devices, locales, and color schemes.
- How to debug any failure with the trace viewer, diagnose and cure flaky tests, and scale with parallelism and sharding.
- How to run across browsers and devices, produce useful reports, and build CI pipelines with GitHub Actions, Docker, and other systems.

How This Book Is Organized

Part I — Foundations (Chapters 1–5) explains what end-to-end testing is, installs Playwright, and covers tests, locators, and actions. **Part II — Writing Reliable Tests** (Chapters 6–10) covers assertions, forms and validation, navigation, dialogs, tabs and frames, files, and the tools that help you write tests faster. **Part III — Test Architecture** (Chapters 11–15) organizes the suite with hooks and tags, authentication state, page objects, fixtures, and test data. **Part IV — API, Network, and Visual Testing** (Chapters 16–20) covers API tests, hybrid tests, network mocking, visual regression, accessibility, time, and emulation. **Part V — Debugging and Scale** (Chapters 21–24) covers debugging, flaky tests, parallelism and sharding, and cross-browser and mobile projects. **Part VI — CI and Production** (Chapters 25–28) covers reporters, CI pipelines, test strategy, and a complete case study. The appendices provide a cheat sheet, templates and a full reference for the Fernway Shop app, a production checklist, and a glossary.

Each chapter includes "Try It Yourself" exercises and ends with a **Key Takeaways** box for quick review.

The Running Example: Fernway Shop

Throughout the book we test **Fernway Shop**, a fictional online store that sells houseplants, pots, garden tools, and potting soil. It has everything a realistic test suite needs to cover: sign-in with customer, admin, and locked accounts; a product catalog with search, filters, and sorting; a cart with quantities, coupons, and shipping rules; a checkout form with validation; order confirmation and history; an admin dashboard with a modal dialog, CSV upload, and CSV download; a help page with an embedded chat frame and a link that opens a new tab; a dark mode; and a JSON REST API. The app is a small Node.js program with no dependencies, included with this book as `fernway-shop.zip`, so you can run every example on your own machine. Its data is small enough to check by hand, and every test result shown in this book was produced by running the tests against it.

Versions and Conventions

Examples use **Playwright Test 1.63** (the version used in this book) with **TypeScript** on **Node.js 22**; Playwright requires Node.js 20 or later. Playwright releases new versions frequently. Most of what you learn here has been stable for many releases, and where a feature is recent, a Note says so. The test runs shown in the book use Chromium; the same tests run in Firefox and WebKit through projects, as Chapter 24 explains. Commands work on Windows, macOS, and Linux; where an environment variable is set differently in PowerShell, both forms are shown.

> **TIP**
>
> A recommended practice or a technique that makes your work easier.

> **NOTE**
>
> Background information, version differences, or a useful aside.

> **WARNING**
>
> A common mistake or something that can cause flaky tests, lost data, or security problems.

Code, file names, commands, and test output appear in `monospace`; long lines are wrapped to fit the page. A caption above each listing names the file it belongs to, such as `tests/cart.spec.ts` or `playwright.config.ts`. Paths in output are shortened for readability.

Contents

Preface

PART I — FOUNDATIONS

1 Why End-to-End Testing and Why Playwright

2 Setting Up: Installing Playwright and Running Your First Test

3 Anatomy of a Test: test, expect, Fixtures, and the Runner

4 Locators: Finding Elements the Way Users Do

5 Actions and Auto-Waiting: Interacting Reliably

PART II — WRITING RELIABLE TESTS

6 Web-First Assertions: Checking What Users See

7 Forms and Validation: Testing Login and Checkout

8 Navigation, Dialogs, New Tabs, and Frames

9 Files: Uploads, Downloads, and the Admin Dashboard

10 Codegen, UI Mode, and Writing Tests Faster

PART III — TEST ARCHITECTURE

11 Organizing a Test Suite: Files, Hooks, Tags, and Annotations

12 Authentication: Storage State, Setup Projects, and Roles

13 Page Object Models That Stay Maintainable

14 Custom Fixtures: Sharing Setup the Playwright Way

15 Test Data: Seeding, Resetting, and Isolation

PART IV — API, NETWORK, AND VISUAL TESTING

16 API Testing with the request Fixture

17 Hybrid Tests: Fast Setup Through the API

18 Network Interception and Mocking

19 Visual Regression Testing with Screenshots

20 Accessibility, Time, and Emulation

PART V — DEBUGGING AND SCALE

21 Debugging Failures: Trace Viewer, Inspector, and Logs

22 Flaky Tests: Causes, Diagnosis, and Cures

23 Parallelism, Sharding, and Speed

24 Cross-Browser and Mobile Testing with Projects

PART VI — CI AND PRODUCTION

25 Reporters and Test Results

26 Running Playwright in CI: GitHub Actions, Docker, and Other Pipelines

27 Test Strategy at Scale: What to Automate and What Not To

28 Case Study: The Complete Fernway Test Suite

APPENDICES

A Playwright Cheat Sheet

B Templates and Reference

C Production Checklist

D Glossary

PART I

Foundations

What end-to-end tests are for and how Playwright makes them reliable, installing Playwright and running the first test against Fernway Shop, the anatomy of a test and the runner, finding elements the way users do, and interacting with pages without sleeps or guesswork.

1 · Why End-to-End Testing and Why Playwright

2 · Setting Up and Your First Test

3 · Anatomy of a Test

4 · Locators

5 · Actions and Auto-Waiting

CHAPTER 1 · PART I — FOUNDATIONS

Why End-to-End Testing and Why Playwright

Before we write a single line of test code, we need a clear picture of what end-to-end tests are for, why they earned a reputation for being slow and flaky, and how Playwright's design removes most of that pain. This chapter also introduces Fernway Shop, the small online plant store we test throughout the book.

What an End-to-End Test Is

An **end-to-end test** (E2E test) checks a complete user journey through a running application, from the outside, the way a real person would experience it. It opens a real browser, loads real pages from a real server, clicks real buttons, and checks what appears on the screen. Nothing in the middle is replaced: the HTML, the CSS, the browser JavaScript, the HTTP requests, the server code, the session cookie, and the in-memory data all take part, just as they do when a customer visits the site.

Here is a journey we will automate many times in this book. Alice, a customer of Fernway Shop, signs in, finds a Monstera Deliciosa, adds one to her cart, adds two terracotta pots, applies the coupon `SPRING10`, and checks out. At the end she sees an order confirmation page with an order number and a total of **$53.10**. A single end-to-end test can walk through that entire story and confirm that every piece cooperates: the search form, the product cards, the cart badge, the coupon logic, the shipping rule, the checkout validation, and the order page.

No other kind of test gives that particular kind of confidence. A unit test can prove that the discount function returns 590 cents for a 5,900-cent subtotal. It cannot prove that the **Apply** button is wired to that function, that the cart page renders the result as `-$5.90`, or that the button is not hidden behind a cookie banner on a narrow screen. End-to-end tests answer the question that matters most to the business and to users: *can a person still do the thing the application exists for?*

That confidence comes at a cost: a running application, a browser, test data, and failures whose cause can be anywhere in the stack. Much of this book is about keeping that cost low.

Where End-to-End Tests Sit Among Other Tests

Teams usually talk about tests in layers. The names vary between companies, but the ideas are consistent. Each layer trades speed and precision against realism.

Level	What it exercises	Typical tools	Strength

Unit	One function or class in isolation	Jest, Vitest, JUnit	Very fast, pinpoints the broken line
Integration	Several modules together, often with a real database or HTTP layer	Same runners plus test containers or HTTP clients	Catches wiring mistakes between modules
Component	One UI component rendered in a browser or a simulated DOM	Testing Library, framework test kits	Fast feedback on UI behavior
End-to-end	The whole deployed app through its real UI or public API	Playwright, Selenium, Cypress	Highest realism, closest to users

A well-known way to picture the balance is the **test pyramid**: many unit tests at the base, fewer integration tests in the middle, and a small number of end-to-end tests at the top. The shape expresses an idea, not a ratio: push each check down to the cheapest layer that can catch the bug, and reserve end-to-end tests for the journeys that only a full system can prove.

Another picture, the **testing trophy**, puts more weight on integration tests that resemble real usage. Both shapes agree that the layers answer different questions and do not replace each other. The right mix depends on your application and its risks; Chapter 27 offers a risk-based way to decide what deserves an end-to-end test.

> **NOTE**
> Playwright is not only a browser UI tool. Its test runner also includes an HTTP client, so the same suite can test a REST API directly (Chapter 16) and combine API calls with UI steps (Chapter 17). Many teams use Playwright for both "end-to-end through the UI" and "end-to-end through the API" tests.

Why End-to-End Tests Got a Bad Reputation

Ask a developer who has maintained a large browser test suite, and you will often hear a mixture of respect and frustration. The frustration usually has the same few sources. Understanding them is the best way to understand why Playwright works the way it does.

Timing problems

A web page is never "done". After the HTML arrives, scripts run, data is fetched, elements are inserted, animations play, and buttons become enabled. A test that clicks a button a few milliseconds too early clicks nothing, or clicks the wrong thing. Older test suites fought this with fixed sleeps: wait two seconds, then click. Sleeps are slow when the page is fast and too short when the page is slow, so the same test passes on a developer laptop and fails on a busy build server. Fernway Shop's product grid, for example, shows `Loading products...` until its API call returns. A test that reads the result count at the wrong moment sees the loading text instead of `8 products`.

Shared state

Browser tests often share more than their authors realize: a logged-in session, a shopping cart, a row in a database, a value in local storage. When test A leaves an item in the cart, test B sees a total it did not expect. When tests run in parallel, the failures appear and disappear depending on which test happened to run first. These failures are hard to reproduce, so people learn to ignore them, and an ignored test is worse than no test.

Brittle selectors

Many suites find elements with long CSS selectors or XPath expressions copied from the browser's developer tools, such as `div.grid > article:nth-child(3) > button`. These selectors describe the page's internal structure, not what the user sees. A designer wraps the grid in another `div` and twenty tests break, even though nothing changed for users.

Hard-to-debug failures

When a browser test fails on a CI server, you often get one line of text: an element was not found, or a timeout expired. You cannot see what the page looked like or which request failed. Developers rerun the job, it passes, and the real bug survives.

Slow feedback

Starting a browser per test and logging in through the UI before every test adds up. A slow suite runs less often and catches bugs later.

None of these problems is a law of nature. Each one has a design answer, and modern tools, Playwright in particular, build those answers into their defaults.

How Playwright's Design Addresses These Problems

Playwright is an open-source browser automation library maintained by Microsoft, first released in 2020. It comes in two layers. The **Playwright library** drives browsers from code. **Playwright Test**, published on npm as `@playwright/test`, is a complete test runner built on top of that library. This book uses Playwright Test with TypeScript, and when we say "Playwright" we usually mean the two together. Here is how the design maps onto the problems above.

Browser contexts: a fresh browser profile for every test

Playwright starts a browser once per worker process and then creates a lightweight **browser context** for each test. A context is like a brand-new browser profile: it has its own cookies, local storage, cache, and permissions, and it is cheap to create and throw away. Two tests that both add a plant to the cart each get their own Fernway session cookie and therefore their own cart. Isolation is the default; sharing state is something you opt into deliberately (Chapter 12 shows how, and where the traps are).

Auto-waiting before every action

Before Playwright clicks, fills, or checks an element, it waits until the element is attached to the page, visible, stable (not moving), able to receive events, and enabled. If the element is not ready yet, Playwright keeps checking until it is, up to a timeout. You write `await button.click()` and Playwright handles the waiting. There are no sleeps in a well-written Playwright test. Chapter 5 explains these "actionability" checks in detail.

Web-first assertions that retry

Checks such as `await expect(locator).toHaveText('8 products')` do not look at the page once and give up. They retry until the condition is true or a timeout (5 seconds by default) expires. The loading text can appear first; the assertion simply waits for the real value. This single idea removes a whole class of timing failures. Chapter 6 covers these assertions in depth.

Locators that describe what users see

Playwright encourages you to find elements by their **role** and **accessible name**, such as "the button named Add to cart" or "the textbox labeled Email", rather than by CSS structure. These locators survive most redesigns, they read like a description of the user's intent, and they quietly check that your page is accessible. Locators are also lazy: they find the element again every time you use them, so a re-rendered element is never "stale". Chapter 4 is devoted to them.

Tracing and debugging tools

Playwright can record a **trace** of a test: every action, a snapshot of the page before and after it, network requests, console messages, and the test source. When a test fails on a build server, you download the trace and replay it step by step on your own machine. UI mode, an inspector, a code generator, and an HTML report complete the toolbox (Chapters 10 and 21).

One API for three browser engines

The same test code runs in **Chromium** (the engine behind Google Chrome and Microsoft Edge), **Firefox**, and **WebKit** (the engine behind Safari). Playwright downloads browser builds that match its version, so everyone on the team and the CI server run the same browsers. It can also drive the branded Chrome and Edge installations and emulate mobile devices with the right viewport, user agent, and touch support. Chapter 24 shows how projects run one suite across browsers and devices.

Parallel by default, with batteries included

Playwright Test runs test files in parallel worker processes out of the box, and it can split a suite across several machines with sharding. The runner also includes fixtures, retries, tags, reporters, screenshot comparison, and a configuration file, so you do not have to assemble a test stack from separate libraries. Chapters 11, 14, 23, and 25 explore these features.

To give you a feel for the result, here is the first test we will write in Chapter 2. Do not worry about the details yet; notice that it reads almost like a description of the home page and contains no waits at all.

tests/smoke.spec.ts

```
import { test, expect } from '@playwright/test';

test('home page loads', async ({ page }) => {
  await page.goto('/');
  await expect(page).toHaveTitle('Fernway Shop');
  await expect(
    page.getByRole('heading', { name: 'Plants and tools for happy homes' }),
  ).toBeVisible();
});
```

The test opens the home page, checks the page title, and checks that the main heading is visible. Behind those three lines, Playwright started a browser, created an isolated context, opened a page, waited for navigation, and retried both checks until the page was ready.

> **TIP**
>
> When a test reads like a short story about a user ("go to the home page, the title is Fernway Shop, the welcome heading is visible"), it is usually a good test. If you cannot describe a test in one sentence, it is probably testing too many things at once.

Playwright, Selenium, and Cypress: A Fair Comparison

Playwright is not the only good tool, and you may already use another one. The honest way to compare tools is by their architecture and design choices, because those choices decide what each tool is naturally good at. We will not quote speed comparisons: the speed of a suite depends far more on how its tests are written than on the tool.

Selenium and WebDriver

Selenium is the long-established standard for browser automation, with bindings for many languages. It speaks the **W3C WebDriver** protocol: your test sends HTTP commands to a driver program (such as ChromeDriver or GeckoDriver), which controls the browser. Because WebDriver is a W3C standard implemented by browser vendors, Selenium works with a very wide range of browsers and with large cloud device grids, and it has a huge ecosystem. Recent Selenium versions also support **WebDriver BiDi**, a newer bidirectional protocol that adds event-driven features. Selenium itself is a browser automation library rather than a full test runner: teams combine it with a runner, an assertion library, and their own waiting and reporting utilities.

Cypress

Cypress is a JavaScript end-to-end testing tool known for its interactive runner and developer experience. Its test code runs inside the browser, in the same run loop as the application, which gives it direct access to the page and makes time travel and DOM inspection natural. That design also shapes its trade-offs: tests are written in JavaScript or TypeScript, and scenarios that span several browser tabs or several origins need extra care. Cypress supports Chrome-family browsers and Firefox, and has offered WebKit support as an experimental feature.

Playwright

Playwright's tests run in a Node.js process outside the browser and control the browser over a single persistent connection, using the Chrome DevTools Protocol for Chromium and comparable protocols for its Firefox and WebKit builds. Because the test lives outside the page, it can open several pages, several browser contexts, and several origins in one test, intercept and mock network traffic, and wait for browser events such as downloads, dialogs, and new tabs. The trade-off is that Playwright relies on browser builds patched and released together with each Playwright version, rather than on any browser a vendor ships.

Aspect	Selenium	Cypress	Playwright
Where tests run	Outside the browser, via a driver	Inside the browser, next to the app	Outside the browser, over one connection
Protocol	W3C WebDriver (and BiDi)	In-browser JavaScript plus a Node.js process	DevTools-style protocols
Languages	Many	JavaScript, TypeScript	TypeScript, JavaScript, Python, Java, .NET
Built-in runner	No (use your own)	Yes	Yes (Node.js)
Auto-waiting	Explicit waits you write	Built into commands	Built into actions and assertions

All three tools are actively maintained. If you already have a healthy suite in another tool, a rewrite is rarely the most valuable work; if you are starting fresh, Playwright's defaults make reliable tests the easy path.

> **NOTE**
>
> Playwright also has official bindings for Python, Java, and .NET. Those bindings share the same browser automation API but use their own test runners (for example pytest or JUnit). This book covers the Node.js test runner, Playwright Test, which has the richest set of runner features such as fixtures, projects, and the built-in reporters.

What Playwright Is Not

Knowing a tool's limits is as important as knowing its strengths. Playwright is not the right tool for the following jobs, even though it may look like it could do them.

- **Load and performance testing.** Each worker drives a real browser that uses real CPU and memory, so you cannot simulate thousands of users that way. Use a dedicated load-testing tool.
- **Unit testing.** Running a browser to test a pure function is slow; keep business logic, such as the coupon rounding rule, covered by unit tests as well.
- **Security scanning.** Playwright can check that a customer gets "Admins only" on the admin page, but it does not hunt for vulnerabilities like a security scanner does.
- **Real-device testing.** Mobile emulation sets the viewport, user agent, device scale factor, and touch support, but it still runs a desktop browser engine. It is excellent for responsive layouts, and it is not the same as a physical phone. Chapter 24 explains the difference.
- **A replacement for human judgment.** Tests confirm what you told them to expect. They do not notice that a new feature is confusing, that a color is ugly, or that a message is rude. Exploratory testing by people remains valuable.

> **WARNING**
>
> Never point a test suite that creates orders, edits products, or resets data at a production site. The tests in this book run against your local copy of Fernway Shop. Chapter 27 explains how to design separate, read-only smoke checks when you do need to verify a live site.

Meet Fernway Shop

Toy pages teach the API but not the craft. Real applications have sign-in, roles, shared data, validation, money, slow requests, dialogs, frames, and files. So this book tests one realistic application from start to finish: **Fernway Shop**, a fictional online store that sells houseplants, pots, garden tools, and potting soil.

Fernway Shop ships with the book as the companion download `fernway-shop.zip`. It contains a `shop/` folder with a small Node.js server (`server.mjs`), its seed data (`data.mjs`), the browser pages (`public/`), and a `README.txt`. It has no dependencies to install: you need only Node.js. All data lives in memory and is restored to the same starting point every time the server starts, which makes it ideal for testing. Chapter 2 walks through starting it.

The pages

Path	Page	What we test there
`/`	Home	Title, welcome heading, three featured products
`/products`	Shop	Search, category filter, sorting, add to cart

`/products/:id`	Product details	Price, stock, quantity, add to cart
`/cart`	Your cart	Quantities, coupons, shipping, totals, clear cart
`/login`	Sign in	Success, wrong password, locked account, redirects
`/checkout`	Checkout	Form validation, payment method, placing orders
`/orders/:number`	Order	Confirmation, order number, total
`/account`	My orders	Order history per customer
`/admin`	Admin dashboard	Add, edit, delete, import, export
`/help`	Help center	FAQs, a new tab, an embedded chat frame

Every page shares a header with links to **Shop**, **Help**, and the **Cart** (with a badge that shows how many items are in the cart), a **Sign in** link for guests, a greeting such as `Hi, Alice` for signed-in users, and a **Dark mode** button. Short messages, such as `Added Monstera Deliciosa to your cart.`, appear in a toast that disappears after three seconds. Behind the pages sits a JSON REST API under `/api`, which we test directly in Part IV.

The people

The shop comes with four fictional accounts. All of them use the reserved `example.com` domain.

Account	Email	Password	Role
Alice Moreno	alice@example.com	Shopper#2026	Customer (our main user)
Bob Lee	bob@example.com	Shopper#2026	Customer with order FW-1001
Dana Whitfield	admin@example.com	Admin#2026	Admin
Lena Locke	locked@example.com	Shopper#2026	Customer, account locked

These passwords are demo values for a local learning app. From Chapter 12 on, our tests read passwords from environment variables and fall back to these demo values only for the local shop, which is the habit you want for real projects (Chapter 15 looks at that habit in more depth).

The products

The catalog is small enough to check by hand, which matters: when a test says the result should be two products, you can verify it yourself.

SKU	Name	Category	Price	Stock
FW-101	Monstera Deliciosa	Plants	$34.00	12
FW-102	Snake Plant Laurentii	Plants	$24.00	20
FW-103	Fiddle Leaf Fig	Plants	$58.00	0 (out of stock)

FW-201	Terracotta Pot 6 in	Pots	$12.50	40
FW-202	Glazed Ceramic Planter	Pots	$29.00	15
FW-301	Copper Watering Can	Tools	$42.00	4 (only 4 left)
FW-302	Pruning Shears	Tools	$19.99	25
FW-401	Organic Potting Mix 10 L	Soil	$14.75	30

Each product card shows a stock label: `Out of stock` when the stock is zero (and the **Add to cart** button is disabled), `Only N left` when five or fewer remain, and `In stock` otherwise. The Fiddle Leaf Fig and the Copper Watering Can give us one example of each special case.

The store's rules

Fernway Shop has a handful of business rules, and they are exactly the kind of logic that breaks quietly when someone refactors a page.

- **Coupons.** `SPRING10` takes 10% off the subtotal, rounded half-up to the cent. `FREESHIP` makes shipping free. Codes are case-insensitive, and a cart can hold one coupon. An unknown code shows `Unknown coupon code.`
- **Shipping.** A flat $5.99 when the subtotal minus the discount is below $50.00, and free at $50.00 or more. The cart shows `Free` when shipping costs nothing. There is no tax.
- **Limits.** At most 10 of each product per cart, and never more than the stock.
- **Orders.** Order numbers look like `FW-1001`, `FW-1002`, and so on. Placing an order reduces stock and empties the cart.

Our main worked example uses these rules. Alice buys one Monstera Deliciosa ($34.00) and two Terracotta Pots (2 × $12.50 = $25.00). The subtotal is **$59.00**. `SPRING10` takes off **$5.90**, leaving $53.10, which is above the $50.00 threshold, so shipping is **Free** and the total is **$53.10**. The cart badge shows `3`. You will see these numbers many times; when one of them changes, a test should fail.

> **TIP**
> Before automating a journey, walk through it by hand once and write down the exact values you expect, as we just did for the $53.10 order. Tests with precise, pre-computed expectations catch far more bugs than tests that only check that "something" appeared.

Fernway Shop also has a few deliberate testing hooks, such as a reset endpoint that restores all data and a `?delay=1500` parameter that slows the product API so we can study waiting.

The Road Ahead

The book is organized in six parts. Each chapter builds on the test project from the previous ones, so by the end you have a complete, professional suite for Fernway Shop that you can use as a template for your own work.

1. **Part I — Foundations** (Chapters 1–5). Install Playwright, write and run tests, learn how the runner works, find elements with locators, and interact with pages reliably.
2. **Part II — Writing Reliable Tests** (Chapters 6–10). Assertions that wait, forms and validation, navigation, dialogs, new tabs, frames, uploads and downloads, and tools that write test code for you.
3. **Part III — Test Architecture** (Chapters 11–15). Organize a growing suite with hooks and tags, log in once with storage state, build page objects and custom fixtures, and manage test data so tests stay independent.
4. **Part IV — API, Network, and Visual Testing** (Chapters 16–20). Test the REST API, speed up UI tests with API setup, mock the network, compare screenshots, check accessibility, control the clock, and emulate devices.
5. **Part V — Debugging and Scale** (Chapters 21–24). Debug failures with traces, cure flaky tests, run in parallel and in shards, and cover several browsers and mobile viewports.
6. **Part VI — CI and Production** (Chapters 25–28). Reporters, continuous integration with GitHub Actions and Docker, test strategy at scale, and a guided tour of the finished suite.

Every chapter is hands-on: we write a test, run it, read the real output, and improve it. Your timings will differ from ours.

What You Need

You need surprisingly little to follow along.

- **A computer** running Windows, macOS, or Linux, with permission to install software.
- **Node.js 20 or later.** Playwright requires it; we use Node.js 22. Chapter 2 shows how to check your version.
- **A code editor.** Any editor works. Visual Studio Code has an official Playwright extension that we use in Chapter 2 and Chapter 10, but it is optional.
- **Basic JavaScript or TypeScript.** Variables, functions, objects, arrays, and `async` / `await`. We explain the TypeScript you need as it appears; you do not need advanced types.
- **Basic HTML and HTTP.** Knowing what a `<button>`, a form, a cookie, and a GET request are is enough.
- **The companion download** `fernway-shop.zip`, and an internet connection the first time you install Playwright and its browsers.

You do not need prior experience with any test framework. We write tests in TypeScript, which Playwright Test runs directly with no build step; Chapter 2 explains how to use plain JavaScript instead if you prefer.

Paths in code captions are relative to the test project folder, `fernway/e2e`, which we create in Chapter 2, and the shop always runs at `http://127.0.0.1:3000`.

A Mindset for Reliable Tests

Tools help, but reliable suites come from a few habits that we will practice in every chapter. It is worth stating them now, so you can recognize them as they appear.

- **Test behavior, not implementation.** Check what a user can see and do: a heading, a button, a total. Avoid checking class names, internal state, or the exact structure of the DOM unless that is what you mean to test.
- **Make every test independent.** A test should pass when it runs alone, first, last, or in parallel with any other test. If it needs data, it creates or selects that data itself.
- **Never wait for a fixed time.** Wait for a condition instead: an element appears, a request finishes, a URL changes. Playwright does most of this for you.
- **Assert precisely.** Prefer `$53.10` over "a total is shown".
- **Treat a flaky test as a bug.** Chapter 22 shows how to find its cause.

Try It Yourself

1. Pick an application you use every day, such as an email client or an online store. Write down three user journeys that you would want an end-to-end test to protect, each in one sentence.
2. For each journey from exercise 1, list one bug that a unit test could catch and one bug that only an end-to-end test could catch.
3. Using the Fernway Shop rules above, compute by hand the total for one Snake Plant Laurentii and one Pruning Shears with no coupon. Then compute it again after changing the Snake Plant quantity to two. (You should get $49.98 and $67.99. Explain why shipping changes.)
4. Compute the total for one Monstera Deliciosa and one Pruning Shears with `SPRING10`. Watch the rounding and the shipping threshold carefully. (The answer is $54.58.)
5. Think of a browser test you have seen fail "for no reason". Which of the causes in this chapter (timing, shared state, brittle selectors, missing debugging information, slow feedback) was the most likely culprit?