CHAPTER 4

Playwright as the Core Engine

Building deterministic execution before adding intelligence

CHAPTER PREMISE An agent can generate, heal, or explore tests only after the execution layer produces trustworthy evidence. Playwright does not make a suite reliable by itself. Its browser isolation, auto-waiting, web-first assertions, and diagnostics give engineering teams the primitives to make a run repeatable, explainable, and worth extending with AI.

WHAT YOU WILL LEARN

- Build a maintainable Playwright foundation that runs across Chromium, Firefox, and WebKit with bounded retries and useful failure evidence.
- Replace timing guesses and fragile selectors with web-first assertions, user-facing locators, controlled state, and independent browser contexts.
- Use UI and API capabilities together while keeping tests deterministic enough for later generation, healing, prioritization, and agentic exploration.

IN THE REFERENCE SYSTEM You stand up the reference application's baseline deterministic Playwright suite. It covers guest checkout, promotion application, order confirmation, and one service-backed assertion. The suite runs across the three major browser engines, creates its own data, retains diagnostic evidence on failure, and deliberately avoids the fragile patterns that would make later AI layers unreliable.

The Test That Needed Four Seconds

A checkout test has become the most discussed line in a release pipeline. Locally it passes. In CI, it fails roughly one out of every twelve runs. The test clicks Apply promotion, waits four seconds, reads the total, and then continues. When it fails, the team increases the wait to six seconds. When it fails again after a browser upgrade, the team adds another wait before the click.

The visible problem is a flaky test. The deeper problem is that nobody can say what the test is actually waiting for. Is the promotion request still in flight? Has the cart total been recomputed? Is the button disabled because the user is not eligible? Did an unrelated analytics call slow the page? A sleep answers none of those questions. It delays the test and makes the passing run feel calmer, but it does not establish that the application reached the state the test intends to verify.

This is where an AI-native testing program must begin: not with a model prompt, but with a deterministic execution contract. The system needs a known starting state, an interaction expressed in user terms, a condition that proves progress, an oracle that defines correctness, and evidence that permits a human or later agent to explain a failure. Without those elements, an LLM simply receives ambiguous traces and unstable outcomes at much greater speed.

Playwright is the worked engine in this book because it combines multi-browser automation, browser-context isolation, auto-waiting behavior, API access, and strong debugging artifacts in one test runner. Those capabilities make it a useful foundation. They are not a substitute for design. A poorly scoped test with shared data, hard waits, unstable locators, and vague assertions will still be poorly scoped when expressed through Playwright.

FIGURE 4.1 | THE DETERMINISTIC EXECUTION LOOP

INTENT	STATE	ACTION	ASSERT	EVIDENCE
A named user outcome and an explicit oracle.	Known data, time, account, and environment.	Playwright performs one user-visible interaction.	Web-first checks wait for observable behavior.	Trace, screenshot, network facts, and result are retained.

A test becomes explainable when each hand-off is deliberate. An LLM can later help at several stages; it should not hide any of them.

The goal is not a mathematically perfect universe in which every external dependency behaves the same way forever. The goal is a controlled test boundary. A test should make its important variables explicit, own the state it changes, wait on conditions rather than elapsed time, and leave behind enough evidence to distinguish an application regression from a test, data, environment, or dependency failure. That is the execution layer later chapters will ask agents to consume and extend.

4.1 Define the Deterministic Execution Contract

A deterministic test is not one that never observes variation. It is one where relevant variation is controlled, modelled, or named well enough that a result is interpretable. The word “deterministic” therefore describes the test system as much as the script.

Source of variation	What makes tests unstable	Execution-layer response	What to retain on failure
UI readiness	The page renders progressively; a control exists before it is actionable.	Use locator actions and web-first assertions that wait for the relevant user-visible condition.	Trace step, DOM snapshot, screenshot, and action location.
Data and account state	A prior run consumed the cart, promotion, email address, or entitlement.	Create unique fixtures; reset only through supported test controls; make each test own its state.	Fixture ID, account or cart ID, seed version, correlation ID.
Time and scheduled work	Expiry windows, retries, cache refreshes, and background jobs cross test boundaries.	Freeze or drive time where the product permits it; wait on a domain outcome, not a guessed delay.	Clock setting, queued-work state, timestamps, and event IDs.
External services	A payment sandbox, recommendation API, analytics tag, or feature flag responds unpredictably.	Use a real integration only when it is part of the decision under test; otherwise control the boundary.	Request and response evidence, mock version, dependency health, and route used.
Browser and parallelism	Shared storage, global accounts, and hidden order dependencies collide under concurrency.	Use an isolated browser context and independent data per test; make parallel execution a design input.	Browser project, worker index, storage state, and retry number.

A run needs a declared boundary

A test does not need to control the entire world. It needs to declare the part of the world that influences the decision it is making. For a guest checkout test, that includes the product price, inventory availability, promotion policy, shipping choice, tax rules relevant to the fixture, and the order API. It usually does not include the behavior of a live recommendation service or a production analytics tag. The test should either remove those unrelated variables from the path or record that they are intentionally in scope.

This boundary is not a documentation formality. It determines which failures the test may legitimately diagnose. A test that depends on a live third-party payment sandbox can still be valuable, but it is an integration signal with a different reliability profile from a deterministic UI regression test. Conflating the two creates false expectations. The first kind needs dependency status and graceful quarantine policy; the second needs a strict failure standard because it is meant to protect a release gate.

Treat every test as a small experiment. The input, state, action, expected outcome, and evidence should be discernible from the test artifact and its fixtures. This is also the first safeguard against AI-generated noise. When a generator later proposes a test, it must be able to find the same boundary and use the same fixture conventions rather than inventing a hidden dependency.

Determinism is a spectrum, not a claim of purity

Some product behavior is intentionally variable. A search ranking may change with a model update. A recommendation pane may return different, valid suggestions. A generative assistant will not emit identical wording on every call. The execution layer should not pretend those features are traditional fixed-output tests. It should make the variable component explicit, control the parts that can be controlled, and pass a well-defined evidence package to the oracle appropriate for that behavior. Chapter 3 established the distinction between acting and judging; the execution layer supplies the reliable observations that judgment requires.

For conventional application paths, however, teams often accept accidental variability too readily. They label a failure “flaky” when the actual issue is shared data, ambiguous readiness, a weak locator, a background request, or a missing state transition. A flake rate is not an unavoidable property of browser automation. It is a measurable sign that the test boundary needs engineering attention.

EXECUTION RULE A test may wait for a condition whose truth it can name: a button becomes enabled, a request completes, an order status changes, or a customer-visible value is rendered. It should not wait for elapsed time merely because the condition is inconvenient to express.

4.2 Set Up a Project for Independent Runs

A clean Playwright setup gives the suite a home for tests, fixtures, support code, and evidence. The layout should make it difficult to hide shared state or invent ad hoc conventions as the suite grows.

Start with the standard Playwright Test runner and TypeScript configuration. The reference system uses one repository for browser tests, API preparation utilities, fixtures, and reporting configuration. That does not mean every test belongs at the UI layer. It means the runner can establish state through supported APIs, drive the browser where experience matters, and produce one coherent evidence trail.

REFERENCE PROJECT LAYOUT

```
tests/
  e2e/
    checkout.spec.ts
    catalog.spec.ts
  fixtures/
    test.ts
    checkout-fixture.ts
  support/
    api-client.ts
    money.ts
    test-data.ts
playwright/
  .auth/                # introduced fully in Chapter 6
  artifacts/
  snapshots/
playwright.config.ts
```

The directory names are less important than the separation of concerns. Test specifications should tell a reader what behavior is being protected. Fixtures should prepare known state and expose only the objects the test needs. Support utilities should represent stable concepts such as money formatting, API access, or approved test-data creation. Generated artifacts should be written

outside the source directories so they are searchable when a run fails but do not contaminate the code review surface.

Configuration is policy, not boilerplate

The Playwright configuration answers operational questions that otherwise get decided inconsistently by each author: how long an action may take, whether a retry is allowed, which browsers matter, what evidence is retained, and how the application is started. Make these choices visible. The configuration becomes a compact statement of the suite's reliability policy.

BASELINE PLAYWRIGHT.CONFIG.TS

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

export default defineConfig({
  testDir: './tests/e2e',
  timeout: 30_000,
  expect: { timeout: 5_000 },
  fullyParallel: true,
  forbidOnly: !!process.env.CI,
  retries: process.env.CI ? 1 : 0,
  workers: process.env.CI ? 4 : undefined,
  reporter: [
    ['html', { open: 'never', outputFolder: 'playwright/artifacts/html-report' }],
    ['junit', { outputFile: 'playwright/artifacts/junit.xml' }],
  ],
  use: {
    baseURL: process.env.BASE_URL ?? 'http://127.0.0.1:3000',
    trace: 'on-first-retry',
    screenshot: 'only-on-failure',
    video: 'retain-on-failure',
  },
  projects: [
    { name: 'chromium', use: { ...devices['Desktop Chrome'] } },
    { name: 'firefox',  use: { ...devices['Desktop Firefox'] } },
    { name: 'webkit',   use: { ...devices['Desktop Safari'] } },
  ],
  webServer: {
    command: 'npm run dev',
    url: 'http://127.0.0.1:3000',
    reuseExistingServer: !process.env.CI,
  },
});
```

Several details deserve attention. A suite timeout is a guardrail, not a waiting strategy. It caps a test that has become stuck; it does not tell the test when an application is ready. The expectation timeout is intentionally shorter because an assertion should fail close to the unmet condition. The browser projects make compatibility visible rather than assumed. The CI retry is deliberately bounded to one: it can collect a trace for diagnosis and reveal a potential intermittent condition, but it must not turn repeated retries into a mechanism for declaring a defect harmless.

The trace policy is equally purposeful. Capturing a full trace for every successful run is often unnecessary overhead. Capturing it on the first retry provides a richer record precisely when the original outcome is disputed. For high-risk journeys, a team may choose an always-on evidence policy in a restricted lane. The decision should be based on risk and cost, not on a general desire to save every artifact forever.

Configuration choice	Recommended baseline	Reasoning	Do not use it to
Test timeout	A finite ceiling appropriate to the journey and environment.	Stops hung tests and exposes stalled flows.	Encode normal product readiness with a large number.
Expectation timeout	Shorter than the test timeout; tuned to expected UI or API behavior.	Keeps failures local to the missing condition.	Mask a slow service or an ambiguous assertion.

Configuration choice	Recommended baseline	Reasoning	Do not use it to
Retry	One CI retry for diagnostic evidence; no automatic trust upgrade.	Separates a potential flake from a one-off infrastructure issue.	Allow a release to pass without a failure classification.
Parallel workers	Enable after data and state isolation are proven.	Reduces feedback time while exposing hidden shared state early.	Run tests that share accounts, carts, queues, or rate limits.
Artifacts	Trace on retry; screenshots and video on failure; structured report always.	Provides a usable record without overwhelming normal runs.	Replace logs, correlation IDs, or ownership metadata.

Cross-browser execution is part of the product contract

A cross-browser configuration does not mean every test must run in every browser on every pull request. It means the organization has made a conscious choice about the browser engines it supports, the evidence it needs before release, and the lane where that evidence is produced. A practical pattern is a fast Chromium path for routine feedback, followed by a scheduled or pre-release matrix across Chromium, Firefox, and WebKit. High-value user journeys and browser-sensitive UI components deserve the fuller matrix more frequently.

The key distinction is between a strategic test matrix and a convenience sample. A single Chromium run confirms that the script can interact with one engine. It does not prove that the rendered experience, browser APIs, fonts, event behavior, or layout assumptions hold in the other engines your customers use. When the matrix finds a difference, record whether it is an application defect, a deliberate support boundary, or a test assumption. Do not normalize the difference away by skipping the failing project.

PRACTITIONER CHECK Before a test is permitted to run in parallel, ask: Can it be executed twice at the same time with no shared account, record, promotion, mailbox, clock, queue, browser storage, or global cleanup? If not, the test is not ready for parallelism - regardless of whether it currently appears fast.

4.3 Use Web-First Execution Instead of Timing Guesses

The most visible reliability benefit of Playwright is its ability to wait for actionability and retry assertions. The subtle benefit is that it encourages a better question: what observable condition proves that the application is ready for this next step?

When you call locator.click(), Playwright resolves the locator and waits for the element to be actionable. When you use a web-first assertion such as toHaveText or toBeVisible, the runner retries the assertion until the expected state is reached or its timeout expires. This replaces a common anti-pattern: inserting a fixed sleep between interactions because the author saw a race once on a particular machine.

REPLACE ELAPSED TIME WITH AN OBSERVABLE CONDITION

```
// Fragile: four seconds is a delay, not a business condition.
await page.getByRole('button', { name: 'Apply promotion' }).click();
await page.waitForTimeout(4_000);
expect(await page.getByTestId('cart-total').textContent()).toContain('$45.00');

// Web-first: the assertion names the outcome that matters.
await page.getByRole('button', { name: 'Apply promotion' }).click();
await expect(page.getByTestId('promotion-status'))
  .toHaveText('Promotion applied');
await expect(page.getByTestId('cart-total'))
  .toHaveText('$45.00');
```

The second form is not merely shorter. It leaves a useful failure. If the promotion is rejected, the status expectation points to the actual contradiction. If the status appears but the total remains wrong, the total assertion provides a distinct signal. If neither appears, the trace shows whether the button was actionable, whether the request occurred, and what state the page reached. The failure can now be classified. The sleep-based test could only report that it looked too early or too late from one machine's perspective.

Actionability is necessary but not sufficient

Auto-waiting protects against certain categories of UI race: a target may be attached to the DOM but still hidden, moving, covered, disabled, or detached during an interaction. That is valuable. It does not know your business semantics. A button can be perfectly actionable while the customer's shipping calculation is still stale. A list can be visible while its data is from the previous account. A heading can render before a background request has finished changing the value you need to verify.

Your test must therefore choose the condition at the right level. Wait for the state transition, not a generic signal that the page exists. In a checkout, that may be the order-confirmation reference; in a file upload, it may be the server-side processing status; in a task workflow, it may be a durable status record rather than a toast. The most robust assertion is usually closest to the user decision or system state the test is protecting.

What the test needs to know	Weak signal	Stronger condition	Example assertion
The page is ready for interaction	A fixed pause after navigation.	The named control is visible and enabled.	await expect(checkoutButton).toBeEnabled();
A server-side action completed	A spinner disappeared.	The relevant response completed or a durable status changed.	await expect(orderStatus).toHaveText("Confirmed");
The correct data is displayed	The section exists.	The user-visible value matches the approved fixture or oracle.	await expect(total).toHaveText(fixture.total);
A list has refreshed	The first row is visible.	A specific record or count associated with the current fixture is present.	await expect(rowFor(orderId)).toBeVisible();
An error is handled	The click did not throw.	The product displays the intended error state and preserves or rolls back data as specified.	await expect(alert).toHaveText(expectedMessage);

Make asynchronous work explicit

Some flows are genuinely asynchronous and cannot be reduced to a single DOM assertion. Order fulfillment may publish an event; a report may be generated in the background; a fraud check may move through pending and approved states. In those cases, model the sequence. Initiate the action, capture a correlation or entity ID, and assert the state transition at a controlled boundary. A polling helper can be appropriate when it reads a real domain status and is bounded by a known SLA. It is not the same as waiting for an arbitrary amount of time and hoping the work completed.

WAIT FOR THE SERVICE BOUNDARY AND THEN VERIFY THE EXPERIENCE

```
const responsePromise = page.waitForResponse((response) =>
  response.url().includes('/api/orders') &&
  response.request().method() === 'POST'
);

await page.getByRole('button', { name: 'Place order' }).click();
const orderResponse = await responsePromise;
expect(orderResponse.ok()).toBeTruthy();

const order = await orderResponse.json();
await expect(page.getByTestId('order-reference'))
  .toHaveText(order.reference);
await expect(page.getByTestId('order-status'))
  .toHaveText('Confirmed');
```

The code above does not make the response itself the complete oracle. For the reference system, the expected price and eligibility rules remain independently defined, as Chapter 3 requires. The response gives the test a correlation point. It connects the user action to the service outcome and lets the test assert that the confirmation page presents the matching order. This is a useful pattern whenever the UI and API cooperate to produce a customer outcome.

COMMON PITFALL Adding waitForTimeout calls whenever a test flakes. The test becomes slower, then still fails under a different load profile, while the meaningful readiness condition remains undocumented.

INSTEAD Name the state transition that must occur and wait on that condition through a locator, response, event-aware status check, or bounded domain polling helper. Use a timeout only as a ceiling for an unmet condition.

4.4 Treat Locators as Product Interfaces

A locator is not an implementation detail. It is the test's claim about how a user or an assistive technology can find the intended element. Weak locators bind tests to styling and structure. Strong locators bind them to behavior and meaning.

The default order of preference is simple. Use an accessible role and accessible name when the interaction is part of the product experience. Use a label for a form control. Use text when the text itself is the behavior being verified. Use a test ID for a stable application contract when the element has no suitable user-facing identity, when the text changes by locale, or when the assertion needs to target a specific state surface. CSS chains and XPath should be treated as last-resort implementation probes, not the primary language of a resilient suite.

Locator approach	What it expresses	Use it for	Why it survives change better
getByRole	A user-recognizable control and its accessible name.	Buttons, links, dialogs, headings, tabs, menu items.	It follows accessibility semantics rather than visual nesting.
getByLabel	The relationship between a form label and its input.	Email fields, promotion input, address forms, search controls.	It remains meaningful when layout or markup around the field changes.
getByText	Visible customer language.	Notifications, status messages, static content, short action labels.	It asserts product copy deliberately, but can be locale-sensitive.
getByTestId	A deliberate test contract owned by the application.	Dynamic totals, repeatable data rows, non-textual visual state, stable integration points.	It avoids dependence on CSS classes and can remain stable across visual redesigns.
CSS / XPath chain	DOM structure or styling choices.	Temporary diagnostics or cases with no semantic alternative.	It usually breaks when developers refactor markup without changing behavior.

Write locators as product language

Consider the difference between page.locator("div.checkout-card button:nth-child(2)") and page.getByRole("button", { name: "Place order" }). The first knows where a button happened to be in a visual container. The second knows what the customer is trying to do. When a designer rearranges the checkout card, the first test may fail even though the product behavior remains intact. The second test still has a useful chance to succeed because its contract is tied to the intended control.

This does not mean every visible string must become a test interface. Product copy changes for legitimate reasons, and localization can make text-based selectors noisy. The discipline is to choose a locator that matches why the test needs the element. A test that verifies a promotion banner's language should use the language deliberately. A test that needs a stable total value across locales should use a semantic test ID and assert the locale-formatted content separately where that behavior matters.

LOCATOR CHOICES THAT COMMUNICATE INTENT

```
// Structural and styling-dependent.
await page.locator('.checkout-panel > div:nth-child(3) button.primary').click();

// User-facing interaction.
await page.getByRole('button', { name: 'Place order' }).click();

// Semantic form relationship.
await page.getByLabel('Promotion code').fill('MEMBER10');

// Stable state surface with an explicit contract.
await expect(page.getByTestId('cart-total')).toHaveText('$45.00');
```

Avoid the false comfort of broad locators

A locator can be semantically named and still be ambiguous. getByText("Continue") is not a reliable contract when several controls on the page share the same word. A broad locator that resolves to the first match may pass until a new design component introduces another matching action. Constrain the locator by role, accessible name, container with a meaningful role, or a dedicated test ID. Strong locators are precise without relying on accidental DOM geometry.

This precision matters later for self-healing. A repair mechanism can only be safely evaluated if the original locator expressed enough intent to distinguish the intended element from a nearby alternative. A vague selector creates a vague healing target. Chapter 6 formalizes stable locator and agent tool contracts; the baseline discipline begins here.

LOCATOR POLICY Every locator should answer one question: what product meaning makes this element the right target? If the answer is “because it is the second button inside this div,” the test has captured layout rather than behavior.

4.5 Isolate Browser State and Test Data

Parallel execution is not a runner setting that can be applied at the end. It is a property earned by isolation. Browser contexts and independent fixtures prevent one run from quietly becoming the precondition for another.

A Playwright browser context is a lightweight, isolated session. Separate contexts isolate cookies, local storage, session storage, permissions, and cache. That is foundational. If one test signs in, changes a preference, or opens a cart, another test should not inherit those artifacts unless the test

explicitly models a collaboration scenario. The runner can create a fresh context per test, while fixtures prepare the data that context will use.

State isolation has two dimensions. Browser isolation is about what the client remembers. Data isolation is about what the product stores. A suite can use separate browser contexts and still collide because each test applies the same promotion code to the same customer, closes the same ticket, or empties the same inventory record. Treat fixtures as first-class code. They should create unique or resettable entities, name the test run that owns them, and dispose of them only through supported test mechanisms.

A FIXTURE PREPARES A KNOWN TEST BOUNDARY

```
import { test as base, expect } from '@playwright/test';
import { createCheckoutFixture } from '../support/test-data';

type CheckoutFixture = {
  sku: string;
  promotionCode: string;
  expectedTotalDisplay: string;
  expectedTotalCents: number;
};

export const test = base.extend<{ checkout: CheckoutFixture }>({
  checkout: async ({ request }, use) => {
    const fixture = await createCheckoutFixture(request, {
      scenario: 'eligible-member-promotion',
      uniqueSuffix: crypto.randomUUID(),
    });
    await use(fixture);
  },
});

export { expect };
```

The fixture does not need to expose every implementation detail. The test needs the product identifiers and expected state required to exercise the flow. It should not need to know how a database row is inserted or how an event is injected. Encapsulating that setup behind a supported test-data API creates a seam that can be secured, audited, and later made available to a test agent without exposing arbitrary infrastructure access.

Use data creation, not data folklore

Many brittle suites rely on named accounts that “usually work”: qa-user-01, premium-customer, customer-with-card. Their hidden history becomes part of every test. One test changes a password, consumes a coupon, creates an order, or leaves a pending workflow. The next test fails and the failure is blamed on a browser. Replace folklore with fixtures that create or claim a known state, record its identifier, and make the dependency visible in the test report.

There are legitimate exceptions. Some end-to-end integrations require a shared sandbox customer or a state that takes too long to create on every run. When that is unavoidable, isolate those tests in a distinct lane, serialize access deliberately, add a health check for the shared asset, and make the dependency explicit in reporting. Do not let an exception quietly become the default model for the suite.

Design choice	Use it when	Benefit	Failure mode to prevent
Fresh browser context per test	Almost always for UI tests.	No cookies, storage, permissions, or cached state leak between tests.	A test passes only because a prior test left the user signed in.
Unique fixture per test	The action mutates customer, order, cart, entitlement, or workflow state.	Tests can run concurrently and failures are attributable.	Retries operate on a partially consumed entity.

Design choice	Use it when	Benefit	Failure mode to prevent
Controlled reusable fixture	Creation cost is material and the state can be reset safely.	Faster setup without uncontrolled history.	A stale shared entity becomes an invisible global dependency.
Serialized integration lane	A real shared external system cannot support concurrent isolation.	The suite retains the signal without pretending it is deterministic.	The same unstable dependency is placed in a critical merge gate.

Authentication is a fixture, not a login journey

The reference system begins this chapter with a guest checkout so the deterministic patterns remain visible. Most real suites will also need authenticated flows. Do not make every test traverse the login page as a prerequisite. It adds a separate dependency, creates unnecessary test time, and turns an authentication failure into noise across unrelated journeys. Chapter 6 establishes a storageState authentication fixture and the access boundaries that agents will require. For now, treat login as its own behavior to test deliberately and authenticated state as a reusable fixture to apply intentionally.

4.6 Combine UI and API Testing Without Blurring Their Roles

The same runner can make service requests, prepare data, observe browser traffic, and verify the rendered experience. Used well, this reduces setup time and makes a failure easier to locate. Used poorly, it duplicates the same business scenario across layers without clarifying what each layer is meant to prove.

The practical rule is: use APIs to create or inspect the state that makes a UI journey meaningful; use the browser to verify the interaction and experience a user would have. The UI test should not become a slow, indirect way to manufacture all data. The API assertion should not become an excuse to skip a critical customer-visible confirmation. Each layer should protect the risk it is best placed to see.

THE UI TEST PROTECTS THE CUSTOMER EXPERIENCE

```
import { test, expect } from '../fixtures/test';

test('an eligible member sees the approved checkout total', async ({
  page, checkout,
}) => {
  await page.goto(`/products/${checkout.sku}`);

  await page.getByRole('button', { name: 'Add to cart' }).click();
  await page.getByRole('link', { name: /cart/i }).click();
  await page.getByLabel('Promotion code').fill(checkout.promotionCode);
  await page.getByRole('button', { name: 'Apply promotion' }).click();

  await expect(page.getByTestId('promotion-status'))
    .toHaveText('Promotion applied');
  await expect(page.getByTestId('cart-total'))
    .toHaveText(checkout.expectedTotalDisplay);
});
```

The fixture provides the specific scenario. The browser test proves that a customer can discover the action, enter the code, receive the status, and see the expected total. In a separate test or in a controlled supplemental assertion, the team can verify the service contract, pricing logic, and event consequences. Chapter 12 develops the shared-scenario pattern across UI, API, contract, and asynchronous workflows. The baseline here is more modest: do not force the browser to carry every responsibility simply because it is visible.

Observe network traffic to join the evidence

There are moments when the browser test benefits from knowing which service response corresponds to the customer action. A checkout submission may create an order, a document upload may initiate processing, or an entitlement action may call a specific authorization endpoint. Capturing the relevant response can provide a correlation ID and reveal whether the failure originated below the UI. It should be a targeted observation, not a practice of asserting every request a page happens to make.

The targeted pattern is to start waiting for the expected response before the action that triggers it, perform the action, validate the response boundary, and then validate the customer-visible result. This avoids a race in which the response arrives before the test starts listening. It also makes a missing or failed response a visible diagnostic rather than a later undefined value in the script.

A FOCUSED BROWSER-TO-SERVICE CORRELATION

```
const orderResponse = page.waitForResponse((response) =>
  response.url().endsWith('/api/orders') &&
  response.request().method() === 'POST'
);

await page.getByRole('button', { name: 'Place order' }).click();
const response = await orderResponse;
expect(response.status()).toBe(201);

const order = await response.json();
await expect(page.getByTestId('order-reference'))
  .toHaveText(order.reference);
```

Keep the oracle independent. The returned order reference is evidence about the transaction, not proof that the promotion calculation was correct. For business-critical logic, compare the observed result with the approved fixture or derived property selected in Chapter 3. The execution layer makes that comparison possible; it does not define the policy by itself.

4.7 Make Failure Evidence a First-Class Output

A reliable suite does not merely produce pass or fail. It produces an evidence package suitable for deciding what failed, who should investigate, and whether the run is safe to repeat. This is the raw material for the failure intelligence in Chapter 11.

Browser traces are particularly useful because they preserve the sequence that a plain screenshot cannot: actions, assertions, navigations, network activity, console entries, source locations, and DOM snapshots at key points. Screenshots show the visual state; video can show a transient interaction or animation; structured reports preserve the test name, browser, timing, retry status, and attachment references. API logs and correlation IDs complete the story when the browser interacts with services.

FIGURE 4.2 | FAILURE EVIDENCE SHOULD CONVERGE, NOT MERELY ACCUMULATE

BROWSER TRACE	UI EVIDENCE	NETWORK FACTS	TEST CONTEXT
Action timeline, DOM snapshots, console entries, and source location.	Screenshot, video when needed, visible error state, and rendered values.	Requests, responses, status codes, correlation IDs, and service timing.	Build, browser, input fixture, retry status, environment, and owner.

A screenshot alone tells a story. A correlated evidence set allows an engineer to verify it.

Evidence must be named and correlated

A folder full of screenshots is less useful than it looks when the artifacts cannot be tied to the environment, fixture, browser, retry, and service transaction. Give the test a stable scenario name. Include the fixture or entity identifier in its attachments. Preserve a correlation ID where the product

provides one. When a failure crosses the UI and a service, record the browser project and the deployment or build version. These small choices make later triage dramatically faster.

ATTACH THE BOUNDARY THAT EXPLAINS THE ASSERTION

```
import { test, expect } from '../fixtures/test';

test('checkout preserves the approved total', async ({ page, checkout }, testInfo) => {
  await testInfo.attach('checkout-fixture.json', {
    body: JSON.stringify(checkout, null, 2),
    contentType: 'application/json',
  });

  await page.goto(`/products/${checkout.sku}`);
  // ... exercise the journey ...

  await expect(page.getByTestId('cart-total'))
    .toHaveText(checkout.expectedTotalDisplay);
});
```

Do not use artifacts as an excuse to avoid clean test output. A well-written failure message should still say what the test expected and what the application returned. Artifacts then help answer why. A useful hierarchy is: name the violated behavior in the assertion, attach the specific fixture and correlation facts, retain trace and visual evidence automatically, and surface a concise classification in the report where it is already known.

A retry is a diagnostic event

When a test fails on its first attempt and passes on a retry, the suite has not established that the original failure was harmless. It has established that behavior differed between attempts. The report should retain both outcomes. The team may classify the event as a transient environment issue, a data collision, a timing defect, a third-party dependency event, or a product race. Repeated retry passes are a queue of engineering work, not evidence that the test may be ignored.

This distinction is especially important before adding AI-based triage. A model can cluster retry patterns and inspect traces, but it cannot recover evidence that was never captured or distinguish a legitimate flake from a hidden intermittent product defect without a defined classification rule. Build the evidence discipline before automating the interpretation.

REPORTING RULE A failure report should let a reviewer answer four questions without rerunning the test: What customer behavior was expected? What actually happened? Which fixture and environment produced the result? What evidence supports the likely failure class?

4.8 Debug From the First Contradiction

A flaky test is not a label to attach after several inconvenient runs. It is a contradiction: the same declared test boundary has produced different outcomes. The first discipline is to investigate the earliest point where the run differs from the expected path, rather than changing the timeout, locator, or retry policy until the result becomes green.

Start from the assertion that failed, then work backward through the evidence to the first unexpected state transition. Did the locator point at the intended element? Did the browser action occur? Did the application issue the expected request? Did the service return the expected response? Did the fixture represent the state the test claims it represents? This sequence keeps the investigation close to facts. It prevents a common and costly reflex: changing several pieces of test code at once and losing the ability to tell which change removed the signal.

The developer-level diagnosis in this chapter is intentionally narrower than the failure intelligence platform in Chapter 11. Here, the objective is to make one test explain itself. Later, the system will use

the same evidence to classify large volumes of failures, track flake rates, and route issues to owners. That future automation is only credible when the underlying tests already record their relevant boundary, state, action, and outcome.

First contradiction observed	Evidence to inspect next	Likely class to consider	Do not "fix" it by
The intended control never becomes actionable.	Trace action timeline, DOM snapshot, overlay state, role/name match, browser console.	Locator drift, accessibility regression, loading defect, or blocked interaction.	Adding a sleep before the click or selecting a less specific element.
The customer action occurs but no expected response follows.	Network log, correlation ID, request payload, feature flag, client console.	Application event handling defect, wrong route, client validation, or test setup mismatch.	Waiting longer without checking whether the request was ever sent.
The response succeeds but the UI shows the wrong state.	Response body, page assertion, cached client state, render timing, browser project.	Presentation mapping defect, stale state, browser-specific behavior, or a real business regression.	Replacing the UI assertion with a response-only check.
The expected value differs from the product result.	Fixture version, oracle source, service response, policy version, exact input facts.	Application defect, stale approved case, incorrect data setup, or policy change requiring review.	Updating the expected value to match the current product output.
The retry passes after an initial failure.	Both attempts, retry trace, timing, worker, fixture identifiers, dependency health.	Intermittent product race, data collision, infrastructure event, or dependency variation.	Marking the test stable because the final job became green.

Use a trace-first review sequence

First, read the assertion without opening the trace. It should state the violated behavior in a way a reviewer can understand: the promotion was not applied, the order confirmation did not appear, or the total differed from the approved fixture. Second, inspect the test input and environment. Confirm that the fixture, browser project, deployment, and retry are the ones you think they are. Third, open the trace around the first divergence: action, page state, request, response, and resulting UI. Fourth, form one specific hypothesis and test it against the evidence rather than editing the script immediately.

This sequence sounds ordinary because it is. Its value is consistency. When a team has no shared diagnostic order, each engineer selects the evidence that supports their first intuition. A front-end engineer begins with the DOM, a service engineer begins with logs, and a test author begins with the locator. The resulting conversation can last longer than the failure itself. A trace-first approach does not replace expertise; it gives each role the same timeline and the same starting facts.

Preserve the before and after of every repair

When you repair a test, record what failed and why the repair is valid. A locator change should state whether the product behavior remained intact and the selector was wrong, or whether the product contract changed. A fixture change should state whether the data setup was stale or the business policy changed. A timeout or retry adjustment should name the specific state transition or dependency profile that justified it. This record is the beginning of a repair history - useful today for code review and later for the self-healing audit trail in Chapter 8.

Do not make a single green rerun the acceptance criterion for a repair. Re-run the affected test repeatedly, exercise the relevant browser matrix, and check whether the repair reduced ambiguity or simply changed timing. Where a defect was real, keep the failing reproduction and move the expected behavior into a permanent regression check. This is how a deterministic suite becomes more trustworthy over time rather than merely less noisy on one build.

DEBUGGING RULE Change the smallest thing that the evidence supports. A deterministic repair explains why the prior run failed and why the revised test preserves the intended behavior. A green rerun without that explanation is only a temporary absence of evidence.

Going Deeper: Control Network, Time, and Non-Deterministic Boundaries

Prerequisite: none. Some applications cannot become reliable merely through better locators and assertions. They also depend on time, network behavior, third-party services, or dynamic content that must be controlled at the correct boundary.

Mock the boundary, not the business rule

Network routing is powerful because it can remove a dependency that is irrelevant to the behavior under test. The discipline is to mock only the boundary that you do not intend to validate in that test. For example, a checkout UI test may route a live recommendation request to a fixed response so personalization does not change the page layout or timing. It should not mock the order service if the purpose of the test is to verify the checkout integration. A test that replaces the component it claims to protect has created a false green path.

CONTROL AN UNRELATED DYNAMIC DEPENDENCY

```
await page.route('**/api/recommendations**', async (route) => {
  await route.fulfill({
    status: 200,
    contentType: 'application/json',
    body: JSON.stringify({ items: [] }),
  });
});

await page.goto(`/products/${checkout.sku}`);
// The checkout journey is now insulated from unrelated dynamic recommendations.
```

Treat time as data when product behavior depends on it

Expiring promotions, scheduled deliveries, cache invalidation, delayed retries, and end-of-day processing are common sources of unstable tests. Where the application can accept an injectable clock or a test-controlled time source, use it. In browser-only cases, Playwright can control page-level time behavior. The key question remains the same: what time condition does the business rule require? A test should move time deliberately to that condition and assert the resulting behavior, not wait in real time for a window to pass.

DRIVE A TIME-DEPENDENT POLICY RATHER THAN WAITING FOR IT

```
await page.clock.install({ time: new Date('2026-06-01T09:00:00Z') });
await page.goto('/promotions/member-week');

await expect(page.getByTestId('promotion-status'))
  .toHaveText('Promotion active');

await page.clock.fastForward('24:00:00');
await expect(page.getByTestId('promotion-status'))
  .toHaveText('Promotion expired');
```

Do not freeze every dependency by default. A meaningful integration lane needs real service behavior and genuine timing characteristics. The purpose of control is to remove accidental variability from a test that is not designed to measure it. As the program matures, give each lane a clear label: deterministic regression, service integration, end-to-end sandbox, performance, resilience, or production shadow. The name tells the reader which dependencies are intentionally real and which are bounded.

COMMON PITFALL Mocking every network call until the UI test is perfectly stable. The suite becomes fast and green while the real workflow quietly breaks at the first service boundary.

INSTEAD Control only the variables outside the decision under test. Keep the intended integration real, observe it explicitly, and place full-system checks in a lane with a reliability expectation appropriate to its dependencies.

Hands-On Lab: Build a Deterministic Checkout Baseline

In this lab, you create the reference system's first reliable browser test. The goal is not maximum coverage. It is a pattern you can reproduce: independent setup, semantic interaction, web-first assertions, evidence retention, and a cross-browser run.

Lab goal

Create a Playwright test that adds a product to a cart, applies an eligible promotion, verifies the approved total, and reaches an order confirmation. The test should run independently in Chromium, Firefox, and WebKit. When it fails, the report should show the fixture, browser, trace, screenshot, and relevant service response.

STEP 1 Install the runner and create the baseline configuration

Initialize a TypeScript Playwright project, retain the generated configuration only as a starting point, and then apply the policy choices from this chapter: explicit timeouts, one diagnostic retry in CI, three browser projects, trace on first retry, screenshots and video on failure, and an HTML report. Point baseURL at the reference application's test environment rather than hard-coding URLs inside tests.

INITIALIZE AND RUN THE BASELINE

```
npm init playwright@latest

# Then run the first smoke test in one browser.
npx playwright test --project=chromium

# Run the cross-browser validation when the baseline is ready.
npx playwright test --project=chromium --project=firefox --project=webkit
```

STEP 2 Create a supported checkout fixture

Use a test-support API, a fixture factory, or another controlled mechanism to create the exact product and promotion state the scenario requires. The fixture must return an identifier, a promotion code, and the approved expected total. Avoid an account or cart that may have been changed by a prior run. Store the fixture data in an attachment so a future failure can be reproduced.

EXAMPLE TEST-SUPPORT FIXTURE BOUNDARY

```
export async function createCheckoutFixture(request, options) {
  const response = await request.post('/test-support/checkout-fixtures', {
    data: {
      scenario: options.scenario,
      uniqueSuffix: options.uniqueSuffix,
    },
  });

  if (!response.ok()) {
    throw new Error(`Fixture creation failed: ${response.status()}`);
  }

  return await response.json();
}
```

STEP 3 Write the customer journey with semantic locators

Navigate to the fixture's product. Add it to the cart through a named button. Enter the promotion in the labelled field. Assert the promotion status and the cart total using web-first assertions. Do not add a sleep. Keep the expected total supplied by the fixture or an independently reviewed oracle, not calculated from the page after the fact.

THE DETERMINISTIC BASELINE TEST

```
test('eligible member receives the approved checkout total', async ({
  page, checkout,
}, testInfo) => {
  await testInfo.attach('fixture.json', {
    body: JSON.stringify(checkout, null, 2),
    contentType: 'application/json',
  });

  await page.goto(`/products/${checkout.sku}`);
  await page.getByRole('button', { name: 'Add to cart' }).click();
  await page.getByRole('link', { name: /cart/i }).click();
  await page.getByLabel('Promotion code').fill(checkout.promotionCode);
  await page.getByRole('button', { name: 'Apply promotion' }).click();

  await expect(page.getByTestId('promotion-status'))
    .toHaveText('Promotion applied');
  await expect(page.getByTestId('cart-total'))
    .toHaveText(checkout.expectedTotalDisplay);
});
```

STEP 4 Join the browser and order-service evidence

Start a targeted wait for the order-creation response before placing the order. After the response returns successfully, assert the order reference on the confirmation page. Preserve the response correlation value in an attachment or structured log if your application supports it. This gives a later triage process evidence across both layers without turning the UI test into a full contract suite.

STEP 5 Run the test repeatedly and then in parallel

Run the test ten or more times in Chromium before trusting a single green result. Then run it in all three browser projects. Finally, run it alongside other isolated tests with multiple workers. Any collision, delay, or browser-specific failure is useful design feedback. Fix the boundary or label the lane correctly; do not suppress the signal with a larger timeout or browser skip.

STEP 6 Inspect one deliberate failure

Temporarily change the expected cart total or disable the promotion response in a test environment. Confirm that the HTML report identifies the failing assertion, the fixture attachment is present, the screenshot captures the relevant UI state, and the trace provides a readable action timeline. Restore the system after the check. A diagnostic mechanism you have never inspected is an assumption, not a capability.

Expected result

You should now have one stable, explanatory checkout scenario. It is deliberately narrow: it does not solve authentication, test generation, self-healing, visual validation, agent control, or complete API coverage. It does create the foundation all of those capabilities depend on. The test has a named user outcome, controlled state, semantic interactions, a machine-checkable expected value, a cross-browser policy, and evidence for failure classification.

Decision Guidance: Choose the Right Execution Boundary

The fastest test is not always the most useful test, and the most realistic test is not always suitable for a critical merge gate. Choose the boundary based on the decision you need the test to protect.

Decision to protect	Prefer this execution boundary	Keep deterministic by	Do not mistake it for
A critical customer journey works in the supported browser experience.	UI journey with controlled fixtures, semantic locators, and explicit customer-facing assertions.	Owning browser and data state; mocking unrelated dynamic dependencies.	Complete proof of every downstream service contract.
A business rule or API contract remains correct across inputs.	API, contract, or property-focused test near the decision layer.	Using approved cases, schemas, derived properties, or reference behavior.	A visual confirmation that the page displayed some value.
A full sandbox integration still works.	A separately labelled end-to-end integration lane.	Recording dependency health and isolating what can be isolated.	A zero-flake release gate with the same expectation as a local deterministic test.
A time-dependent policy behaves correctly.	Test with a controlled clock or injectable time source.	Moving time deliberately to policy boundaries.	A real-time wait that happens to pass today.
A dynamic third-party component does not distract from the workflow under test.	Route or mock the unrelated boundary in a deterministic lane.	Versioning the mock and documenting what is intentionally excluded.	Evidence that the real third-party integration is healthy.

Chapter Checklist

- Replace every unexplained fixed wait in one selected suite with a named readiness condition, response boundary, or bounded domain-status check.
- Configure traces, screenshots, and video so a first-attempt failure and its diagnostic retry can be reviewed without rerunning the pipeline.
- Audit the locators in one critical journey. Replace CSS or XPath chains with role, label, text, or deliberate test-ID contracts where product meaning supports them.
- Prove that one high-value test can run twice concurrently with independent browser state and data. Record what you changed to remove shared state.
- Run the baseline scenario across Chromium, Firefox, and WebKit. Classify any difference rather than skipping the affected browser project.
- For one UI test, state explicitly what the browser journey proves, what the service-level assertion proves, and which oracle remains responsible for correctness.

Exercises and Extensions

Use these short exercises to turn the chapter patterns into evidence in your own codebase. Do not optimise for the number of tests added; optimise for whether each result becomes more repeatable and easier to explain.

- Choose one test that currently relies on a named shared user. Replace it with a unique fixture or an explicitly controlled reusable fixture, then run it with two workers to prove the isolation.
- Take one flaky checkout or workflow test and write down the earliest contradictory observation from its trace. Replace only the unsupported wait, locator, or setup step that evidence justifies.
- Add one failure attachment that a person can use to reproduce the test boundary: fixture JSON, generated entity ID, policy version, or a correlation value.
- Run the same user journey through a fast browser lane and a cross-browser lane. Record which assertions protect customer experience and which need to move to a lower API or contract layer.
- Identify one time- or third-party-dependent path. Decide whether it should be controlled in the deterministic lane, retained in a labelled integration lane, or tested in both with different expectations.

KEY TAKEAWAY AI cannot turn unstable execution into trustworthy testing. A deterministic Playwright foundation controls state, expresses interaction through product semantics, waits for observable conditions, separates UI from service responsibilities, and leaves behind evidence that can be explained and acted on.

Leads into -> Chapter 5 decides which tests deserve to exist and where this deterministic baseline fits in the wider coverage strategy. Chapter 6 then turns stable locators, authenticated contexts, and safe agent tools into an explicit contract that generated and autonomous tests can use.

FURTHER READING

See Appendix H for source notes on Playwright Test, browser isolation, auto-waiting and web-first assertions, locator strategy, tracing, and test-data isolation. Keep fast-changing runner and browser guidance in the maintained online companion described in Appendix G.