Testing JavaScript Applications

Flavio Copes

Updated August 6, 2026

Contents

Test JavaScript from pure functions to APIs, databases, browsers, CI, and small load checks with Node.js, Testcontainers, Playwright, and k6.

What you'll learn: Build a fast, isolated, diagnosable test strategy for the Books application across unit, integration, browser, and reliability boundaries.

Testing foundations

Choose evidence from risk, use the Node test runner, and make tests clear and deterministic.

Choose what a test proves

Start from one risk and one observable behavior instead of writing tests merely to increase a coverage number.

A test is useful when it can fail for a meaningful reason. Begin with a behavior that would hurt a user or developer if it broke.

Use small unit tests for deterministic logic, integration tests where real components meet, and browser tests for a few critical user journeys. The labels matter less than knowing which boundary each test crosses and which failure it can diagnose.

Start with a sentence: "Given this state, when this action happens, then this observable result follows." The last part is the test's oracle: the evidence that tells you whether the behavior is correct. If the oracle only checks that no exception was thrown, a route could return the wrong book and the test would still pass.

For example, the risk "a duplicate ISBN replaces an existing book" crosses the database boundary. A repository integration test using the real uniqueness constraint is stronger than a unit test that teaches a fake repository to reject duplicates. The risk "the title is trimmed" belongs in a fast unit test. The risk "a reader can add a book from the page" deserves one browser test because labels, JavaScript, HTTP, and persistence must cooperate.

Choosing too low a boundary creates false confidence: the pieces pass while their wiring is broken. Choosing too high a boundary creates vague, slow failures: a browser test says "Add book failed" without identifying whether validation or storage caused it. Use the cheapest boundary that can actually expose the risk, then add a broader test only when integration itself is part of the risk.

List five risks in the Books API. For each one, write the observable failure, the boundary that must be real, and the cheapest test level that can prove it. If you cannot name the evidence, the test idea is still too vague.

Run the first Node test

Use the built-in Node.js test runner and strict assertions to execute one fast deterministic test without another framework.

Modern Node.js includes a stable test runner. It discovers tests, reports failures, supports filtering, mocking, and coverage, and works with ordinary modules.

Keep tests beside a small module or under a predictable tests directory. Use strict assertions so coercion cannot turn the wrong type into an apparent pass.

```
import test from 'node:test'
import assert from 'node:assert/strict'
import { normalizeTitle } from './books.js'

test('normalizes surrounding whitespace', () => {
  assert.equal(normalizeTitle('  Dune  '), 'Dune')
})
```

The test name should describe the behavior, not the function name. When this fails, Node reports the test, expected value, actual value, and stack location. That evidence is part of the design: `normalizes title` is less useful than `removes surrounding whitespace but preserves internal spaces`.

Make sure the assertion observes the returned value. This test would be a false positive if it asserted the input string or a constant that the function never produced. A useful habit is to mutate the implementation briefly—return the original string—and confirm that the test turns red for the expected reason. Then restore the implementation and watch it pass again. This red-green check proves that the test is connected to the behavior.

Add a second case for `'The  Left Hand of Darkness'`. Decide whether two internal spaces are preserved or collapsed, encode that decision in the name and expectation, then run `node --test`. Deliberately make one expectation wrong and use the diagnostic to locate the mismatch before correcting it.

Arrange, act, and assert

Structure each example so its setup, one meaningful action, and expected observable result are easy to read.

Tests are executable explanations. A reader should see the situation, action, and consequence without reverse-engineering a large fixture.

Arrange the minimum relevant state, perform the behavior, then assert the result. Several assertions are fine when they describe one outcome. Split a test when it performs unrelated actions or could fail for unrelated reasons.

```
test('rejects a book without an author', () => {
  // Arrange
  const input = { title: 'Dune', author: '  ' }

  // Act
  const result = validateBook(input)

  // Assert
  assert.deepEqual(result, {
    ok: false,
    field: 'author',
    reason: 'required'
  })
})
```

The important separation is conceptual, not the comments. Arrange explains why this situation matters. Act contains one behavior-changing operation. Assert checks what a caller can observe. If the assertion checks only that `input.author` is blank, it proves the setup, not `validateBook()`.

Avoid hiding important behavior in shared setup. A global fixture containing ten books may make every test shorter, but it also makes the cause of a failure harder to see. Keep common setup for genuinely irrelevant details and put risk-relevant values in the test.

Multiple assertions belong together when they describe one contract, such as the status, header, and body of one HTTP response. Split them when one assertion can fail because creating a book

broke and another because deleting a different book broke.

Rewrite one broad Books API test into focused create, reject-invalid-input, and missing-book cases. For each case, underline the single action and verify that every assertion observes its result rather than restating the arrangement.

Make tests deterministic

Control time, randomness, data, environment, and cleanup so the same code produces the same result on every run.

A flaky test sometimes passes and sometimes fails without a product change. It trains the team to ignore the suite.

Inject clocks and ID generators, use isolated data, wait for observable conditions instead of sleeping, and restore modified globals after each case. Parallel execution is safe only when tests do not share mutable state.

Think of determinism as explicit inputs plus controlled dependencies. If a result depends on the wall clock, random UUIDs, process environment, test order, or a row left by another test, you have hidden inputs.

```
const book = createBook(
  { title: 'Dune', author: 'Frank Herbert' },
  {
    now: () => new Date('2026-01-15T10:00:00Z'),
    createId: () => 'book-123'
  }
)

assert.equal(book.id, 'book-123')
assert.equal(book.createdAt, '2026-01-15T10:00:00.000Z')
```

This test can assert exact output without guessing an acceptable time range. A range such as "created within the last second" can pass with the wrong timestamp and fail on a busy CI runner. Likewise, sleeping for 100 milliseconds does not synchronize anything; it only hopes the other work finishes first.

Control randomness with an injected generator or a recorded seed. Give parallel tests unique files, ports, schemas, or record identifiers. Register cleanup as soon as a resource is created so it still runs after an assertion fails. If you replace a global, restore it in teardown; otherwise a passing test can poison the next one.

Change book creation to accept an injected clock and ID generator, then assert exact output without patching global time. Run the test repeatedly and in parallel with another case that uses different fixed values.

Check your understanding: testing foundations

Check test purpose, boundaries, Node test setup, arrange-act-assert, behavior-focused assertions, and test isolation.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Unit tests

Test pure domain logic, boundaries, injected dependencies, and focused test doubles.

Extract pure domain logic

Move validation and transformation rules into small functions whose outputs depend only on explicit inputs.

Pure logic is cheap to test because it does not need a server, database, clock, network, or cleanup.

Extract book normalization and validation from the route handler. Return a value or a typed result instead of mutating global state. Keep HTTP status selection in the HTTP layer.

```
export function normalizeBook(input) {
  const title = input.title?.trim() ?? ''
  const author = input.author?.trim() ?? ''
  const year = input.year === '' || input.year == null
    ? null
    : Number(input.year)

  if (!title) return { ok: false, field: 'title', reason: 'required' }
  if (!author) return { ok: false, field: 'author', reason: 'required' }
  if (year !== null && !Number.isInteger(year)) {
    return { ok: false, field: 'year', reason: 'integer' }
  }

  return { ok: true, value: { title, author, year } }
}
```

This function owns a domain decision: what counts as a valid book value. The HTTP handler owns a transport decision: turning a failed result into a `400` JSON response. Keeping those decisions separate prevents a unit test from needing to construct a `Request`, and prevents the domain function from knowing about Hono.

Purity is not a goal by itself. Do not extract a one-line wrapper merely to mock it. Extract logic when explicit inputs and outputs make an important rule easier to see. The remaining route still needs an integration test, because a perfectly tested normalizer does not prove that the handler calls it or serializes its result correctly.

Create and test a function that trims title and author, validates length, and normalizes an optional year. Include one test that proves the input object is not mutated and one HTTP test that proves a validation failure becomes the documented response.

Test boundaries with tables

Cover a family of related edge cases with data-driven tests while preserving useful case names and diagnostics.

Validation bugs live near boundaries: empty versus one character, maximum length versus one over, and valid versus impossible years.

A table keeps the setup consistent and makes missing cases visible. Give each row a descriptive name; do not hide an entire product specification in a clever loop.

```
const cases = [
  {
    name: 'rejects an empty title',
    input: { title: '', author: 'A' },
    field: 'title'
  },
  {
    name: 'rejects a blank author',
    input: { title: 'Dune', author: '  ' },
    field: 'author'
  }
]

for (const { name, input, field } of cases) {
  test(name, () => {
    const result = validateBook(input)
    assert.equal(result.ok, false)
    assert.equal(result.field, field)
  })
}
```

A boundary is where behavior changes. If titles allow 1 through 200 characters, the informative values are 0, 1, 200, and 201—not an arbitrary middle value such as 73. For a publication year, also ask whether the rule is inclusive, whether numeric strings are accepted, and what happens to `null`, an empty string, a decimal, or `NaN`.

Name every row so the failure identifies the missing rule. A single test containing a loop can stop at the first bad row and hide later failures. Creating one test per row preserves independent diagnostics while keeping the cases together.

Tables work best when all rows exercise the same decision. If one row tests title validation and another tests a repository timeout, their setup and expected evidence are different; separate tests will be clearer. Also include at least one valid row. A validator that rejects every input can make an all-invalid table look correct.

Add valid minimum and maximum values plus one invalid value on each side of every boundary. Temporarily change `<=` to `<` in the implementation and confirm that a boundary case, not a generic example, catches the mistake.

Inject dependencies

Pass repositories, clocks, and ID generators into code so tests can control behavior without patching module internals.

Code is difficult to test when it creates every dependency for itself. Dependency injection can be as simple as a function parameter.

Construct the production handler with the real repository and production clock. Construct the test handler with small deterministic substitutes. Depend on the narrow behavior the handler needs, not an entire database client.

```
export function makeCreateBook({ repository, clock, createId }) {
  return async function createBook(input) {
    const book = {
      ...input,
      id: createId(),
      createdAt: clock.now().toISOString()
    }

    await repository.insert(book)
    return book
  }
}
```

Production composition happens once, near the application entry point. Tests can supply a repository with only `insert()`, a clock with only `now()`, and an ID function. That narrow contract makes the real dependency obvious and prevents test code from imitating an entire database client.

Injection becomes noise when every pure helper is passed through several layers. Use it at boundaries where behavior is slow, nondeterministic, or outside the process: storage, time, IDs, mail, and network calls. Keep domain rules as ordinary functions.

A substitute must also exercise failure behavior. A fake that always succeeds can prove the happy path but cannot prove that a duplicate record is translated correctly. Configure `insert()` to throw the same application-level error the real repository exposes, then assert the caller's observable result. Do not assert private call order unless that order is itself required; otherwise a harmless refactor breaks the test.

Change `createBook` to receive `{ repository, clock, createId }` and test it with in-memory substitutes. Cover one success and one repository failure, then verify that production construction still passes the real implementations.

Use test doubles sparingly

Choose fakes, stubs, spies, and mocks for a precise reason without replacing the behavior the test is supposed to prove.

A fake has working simplified behavior, a stub returns a prepared value, and a spy records interaction. "Mock" is often used for all three.

Prefer a small in-memory repository fake when state matters. Use a stub to trigger a rare error and a spy when an interaction is itself the contract. Reset doubles after tests and avoid mocking your own simple pure functions.

The danger is contract drift. If the real repository returns `null` for a missing book but the fake returns `undefined`, tests can pass against behavior production never provides. Keep doubles behind a narrow application-owned interface and run separate integration tests against the real adapter.

```
const repository = {
  insert: async () => {
    throw new DuplicateIsbnError('9780441172719')
  }
}
```

```
const result = await createBook(input, { repository })
assert.deepEqual(result, {
  ok: false,
  code: 'duplicate-isbn'
})
```

This stub is valuable because it makes a rare failure deterministic. A spy would be appropriate if the contract were "send one confirmation after the transaction commits." It would be weak evidence for "the book was persisted," because observing a call does not prove the database accepted it.

Over-mocking creates false positives by copying the implementation into the test: the test expects the exact internal calls it was taught to expect. Prefer asserting returned values, state changes, or public messages. Spy on interactions only at an external boundary where the interaction is the observable outcome.

Test that a storage conflict becomes the documented API conflict response without connecting to a database. Then add one real-database integration test that proves the configured uniqueness constraint produces the error your stub represents.

Check your understanding: unit tests

Check pure functions, boundaries, table tests, dependency injection, test doubles, error assertions, and implementation coupling.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Integration tests

Test the HTTP stack, real schemas, migrations, SQLite, and disposable PostgreSQL.

Test the HTTP contract

Send real Request objects through the complete Hono routing and middleware stack and assert the public response contract.

Integration at the application boundary catches disagreements among routing, validation, domain logic, serialization, and error handling.

Create the app with injected dependencies, call its request interface, and assert status, headers, and parsed response. Include authentication and error middleware when those are part of the production stack.

```
test('creates a book through the public API', async () => {
  const app = makeApp({ repository: new MemoryBooks() })

  const response = await app.request('/books', {
    method: 'POST',
    headers: { 'content-type': 'application/json' },
    body: JSON.stringify({ title: 'Dune', author: 'Frank Herbert' })
  })
```

```
  assert.equal(response.status, 201)
  assert.match(response.headers.get('content-type'), /^application\/json/)
  assert.deepEqual(await response.json(), {
    id: 'book-1',
    title: 'Dune',
    author: 'Frank Herbert'
  })
})
```

Sending a `Request` through the app proves more than calling the handler directly. It includes route matching, body parsing, middleware order, error translation, and serialization. A direct handler test can pass while production rejects the content type or an authentication middleware prevents the request.

Assert the public contract, not the implementation. Status, relevant headers, and response shape matter. A private repository method name does not. For error cases, assert a stable machine-readable code as well as the status; exact prose often changes without breaking clients.

Beware the opposite false positive: an in-memory repository cannot prove SQL, migration, or driver behavior. Keep this test focused on the application boundary and add repository integration tests for the storage boundary.

Test create then read, invalid JSON, unsupported content type, unknown route, and a storage exception. For each failure, write down which layer should produce the response and make sure the request actually crosses that layer.

Isolate database tests

Give every database test a known schema and independent state so order and parallel execution cannot change the result.

Database tests become flaky when they share rows, depend on an earlier migration test, or leave transactions open.

Create a temporary SQLite database per test or suite, apply migrations from the same files used in production, seed only required data, and close and remove it during cleanup. A transaction rollback strategy is useful only when the code under test shares that transaction.

Isolation means each test can run first, last, alone, or in parallel and produce the same result. Generate a unique temporary path before opening SQLite, register cleanup immediately, and apply the real migration files. Creating tables with test-only SQL can hide a broken production migration.

A transaction around each test is fast, but it has a boundary. If application code opens another connection, commits independently, or runs work after the request returns, the outer rollback may not contain those writes. A fresh database is slower but gives stronger isolation and catches connection behavior. Choose based on what the test must prove.

Seed only the rows named by the scenario. Reusing a large fixture makes uniqueness and ordering bugs hard to diagnose. Avoid fixed IDs across parallel tests unless every test has its own database. Close statements and connections before deleting a temporary file, and make cleanup run even when setup or an assertion fails.

A CRUD sequence proves useful integration only if it checks persisted state. Reading the object returned by `insert()` can pass without a write. Create it through one call, then read it through a

separate repository or HTTP call that queries the database.

Run the complete Books API CRUD sequence against a fresh temporary database. Repeat it twice in parallel with different titles and prove neither test can see the other test's rows.

Test migrations and constraints

Verify that a clean database reaches the current schema and that important constraints reject invalid or conflicting writes.

Application code and schema migrations ship together but often fail separately. A typo in a migration can make every handler unusable.

Apply all migrations to an empty database, inspect the resulting schema, then run representative reads and writes. Also test upgrading from a supported previous schema when production already has data.

A clean-install test and an upgrade test answer different questions. The clean test proves that a new environment can reach the current schema from nothing. The upgrade test proves that an existing environment can move from a supported old version without losing or corrupting data. Running only the clean path misses changes that fail when a table already contains rows.

For an upgrade test, create the previous schema from a frozen migration snapshot, insert representative old data, then apply only the newer migrations. Read the old row through the current repository and assert both preserved values and intentional defaults. Do not construct the "old" database using today's schema helpers; that would reproduce the destination rather than the real starting point.

Constraints are executable guarantees. Test a successful write at the boundary and a failing write beyond it: a duplicate ISBN, a missing required column, or a broken foreign key. Assert the database rejects the write and leaves existing data unchanged. An application validation test alone cannot prove the constraint exists, while a constraint test alone cannot prove the API reports a useful error.

Migration tests should fail forward. Avoid quietly editing a migration already applied in production; add a new migration and test the supported upgrade path.

Add a uniqueness rule, test a clean migration, a duplicate write, and an upgrade containing existing rows. After the rejected duplicate, query the table and prove that exactly one original row remains.

Use Testcontainers for PostgreSQL

Run database integration tests against a disposable real PostgreSQL container when SQLite no longer represents production behavior.

Testcontainers starts real service containers from test code and returns connection details for the randomly allocated instance.

Start PostgreSQL once per appropriate test scope, wait for readiness, apply migrations, and always stop the container in teardown. Pin the image version used in CI. This is slower than a fake but catches driver and database-specific behavior.

Use PostgreSQL when production behavior depends on PostgreSQL: parameter binding, JSON types, case sensitivity, transactions, indexes, or SQL syntax. SQLite is not a smaller PostgreSQL. A green SQLite test can be a false positive if the production query or constraint behaves differently.

```
let container
```

```
try {
  container = await new PostgreSqlContainer('postgres:17-alpine').start()
  const repository = await connectBooksRepository(container.getConnectionUri())
  await migrate(repository)

  await repository.insert({ title: 'Dune', author: 'Frank Herbert' })
  assert.equal((await repository.findByTitle('Dune')).author, 'Frank Herbert')
} finally {
  await container?.stop()
}
```

Pin a version deliberately so a new image does not change CI underneath you. The container being started is not always the same as the service being ready; use the module's readiness behavior or an explicit strategy that represents a usable database. Apply the same migrations and driver configuration as production.

One container per test gives strong isolation but is expensive. One per suite is often a better boundary if every case gets a unique database or schema and cleanup is verified. Never share fixed rows between concurrently running cases. When a failure occurs, retain the query and container logs that distinguish a product defect from an unavailable container runtime.

Write a small repository test that starts PostgreSQL, inserts a book, reads it through a new query, and stops the container even after failure. Then add one PostgreSQL-specific constraint case that the in-memory fake cannot prove.

Check your understanding: integration tests

Check HTTP handler integration, database isolation, migrations, Testcontainers, cleanup, and testing realistic failure behavior.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Browser tests

Exercise user journeys with Playwright locators, web-first assertions, and traces.

Install Playwright

Add Playwright Test, browser binaries, configuration, and one smoke test to the Books interface project.

Browser tests cover the integrated user journey: page, browser behavior, JavaScript, API, and storage.

Use the official initializer in an existing project and choose TypeScript. Configure a local web server command and a base URL so tests navigate to relative paths. Start with Chromium; add browsers when cross-browser risk justifies the time.

```
npm init playwright@latest
npx playwright test
```

```
npx playwright test --ui
```

Keep the server lifecycle in the configuration so local and CI runs use the same entry point:

```
import { defineConfig } from '@playwright/test'

export default defineConfig({
  use: { baseURL: 'http://127.0.0.1:3000' },
  webServer: {
    command: 'npm run start',
    url: 'http://127.0.0.1:3000',
    reuseExistingServer: !process.env.CI
  }
})
```

`webServer.url` is the readiness target. `baseURL` lets the test navigate to `/books` without copying an environment-specific host. In CI, avoid reusing an unknown process; the test command should own the server it exercises.

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

test('shows the books page', async ({ page }) => {
  await page.goto('/books')
  await expect(page.getByRole('heading', { name: 'Books' })).toBeVisible()
})
```

This is intentionally a smoke test. It proves that the application starts, the route loads, and the expected page reaches the accessibility tree. It does not prove creation, persistence, or error handling. Add those journeys only when they protect important user risks; browser suites become slow and vague when they repeat every unit-level edge case.

Create the smoke test, stop the application server, and run it once to confirm Playwright starts and waits for the configured server. Then change the heading expectation and inspect the failure before restoring it.

Locate elements by role

Select controls through accessible roles, names, labels, and visible behavior instead of fragile CSS implementation details.

A locator should resemble how a user or assistive technology finds the element.

Prefer `getByRole`, `getByLabel`, and `getByText`. Use a test ID only when there is no meaningful user-facing selector. Role-based locators also reveal missing labels and ambiguous control names.

```
await page.getByLabel('Title').fill('Dune')
await page.getByLabel('Author').fill('Frank Herbert')
await page.getByRole('button', { name: 'Add book' }).click()
await expect(page.getByRole('listitem')).toContainText('Dune')
```

The accessible name matters as much as the role. A button may get its name from its text, an `aria-label`, or another labelled element. `getByRole('button', { name: 'Add book' })` therefore follows the same semantic surface that keyboard and assistive-technology users depend on.

Locators are strict for actions: if two buttons match, Playwright reports the ambiguity instead

of guessing. Treat that as evidence. Narrow the locator to the relevant form or give the controls distinct accessible names. Reaching for `.first()` can create a false positive by clicking whichever duplicate happens to render first.

CSS classes such as `.btn-primary:nth-child(2)` describe implementation details. A redesign can break the test while the user journey still works. A very broad text match has the opposite risk: it can find "Dune" in a hidden template, navigation item, or unrelated notification and pass before the saved book appears. Scope the assertion to the book list and include the author when that identifies the created record.

Test IDs are appropriate for behavior with no user-facing semantics, but they should be a deliberate stable contract. Do not add inaccessible markup merely to make the test easy.

Test the add-book journey using labels, roles, and visible text. Duplicate the "Add book" button on purpose, read the strictness error, then fix the interface names or locator scope without using `.first()`.

Wait for observable state

Use Playwright actionability and web-first assertions instead of arbitrary sleeps and race-prone immediate checks.

A fixed delay is simultaneously too slow on a fast run and too short on a slow run.

Playwright waits before actions and retries async assertions until their condition is met or the timeout expires. Wait for the UI state the user needs, such as a saved book appearing or a button becoming enabled.

```
await page.getByRole('button', { name: 'Add book' }).click()

const list = page.getByRole('list', { name: 'Books' })
await expect(list.getByRole('listitem')).toContainText('Dune')
```

Before the click, Playwright checks that the locator resolves to one element and that the element is visible, stable, enabled, and able to receive the action. After the click, the web-first assertion repeatedly reads the current page until the expected state appears or the timeout expires. Keeping the locator itself in the assertion is important; reading text once and asserting the resulting string removes that retry behavior.

Wait for the consequence that matters to the user. If saving a book should add a list item, the list item is better evidence than merely waiting for the POST response. A `200` response can still be followed by broken rendering. Wait for a response when the response itself is the contract or when it provides necessary diagnostic evidence, not as a substitute for the visible outcome.

Avoid `waitForTimeout()` and forced actions as routine fixes. They can hide an overlay, disabled control, or missing state transition. When an assertion times out, inspect which condition never became true and whether the application exposed a stable observable state.

Remove every explicit timeout from one test and replace it with locators and assertions about visible state. Slow the API response locally and confirm the test still passes, then prevent the UI update and confirm the assertion fails with evidence about the missing state.

Debug with traces

Use Playwright traces, screenshots, video, and reports to reconstruct a browser failure before adding retries.

A CI browser failure needs evidence because you cannot inspect the original page after the process exits.

Retain a trace on the first retry or failure. The trace viewer shows actions, DOM snapshots, network requests, console output, and timing. Screenshots are useful, but a trace explains how the page reached that state.

Configure traces so routine successful runs stay cheap while the first retry preserves the failed attempt:

```
import { defineConfig } from '@playwright/test'

export default defineConfig({
  retries: process.env.CI ? 1 : 0,
  use: { trace: 'on-first-retry' }
})
```

A retry is diagnostic evidence, not a fix. If the first attempt fails and the retry passes, the run has revealed timing, shared-state, or environment sensitivity. Track that event instead of treating the test as healthy forever.

Read a trace in order. Find the last successful user action, inspect the locator and actionability state of the next one, then compare the DOM snapshot before and after. Check failed or slow network requests and browser console errors. This separates "the button could not be clicked" from "the click worked but the API failed" and "the API worked but the page never rendered the result."

A screenshot captures one visual moment. It cannot show a request that returned `500`, a redirect chain, or an element that was replaced between two actions. Video helps with motion and ordering, while the HTML report groups failures and attachments. Retain the smallest artifact set that answers likely questions, and avoid publishing traces that contain secrets or private test data.

Break a selector deliberately, run with tracing, and open the trace to find the last successful action and current DOM. Then create a server-side `500` and compare the network and console evidence so you can explain why the two failures need different fixes.

Check your understanding: browser tests

Check Playwright installation, user-facing locators, web-first assertions, isolated data, traces, screenshots, and debugging without fixed waits.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Reliability and CI

Use coverage, CI stages, k6 thresholds, and a maintainable test ownership model.

Use coverage as a map

Read line, branch, and function coverage to find unexamined code without treating one percentage as proof of quality.

Coverage says which code executed during tests. It does not say that assertions were useful or requirements were correct.

Branch coverage is valuable for validation and error handling because one executed line can contain several outcomes. Inspect uncovered areas, add tests for meaningful risk, and exclude generated code only with a documented reason.

```
node --test --experimental-test-coverage
```

Suppose a route contains this decision:

```
if (!book) return json({ code: 'not-found' }, 404)
return json(book, 200)
```

A happy-path test may execute the function and most of its lines while never proving the missing-book branch. Branch coverage points to that gap. The next question is not "How do I color the line green?" but "What user-visible failure is untested?" Send a request for an absent ID and assert the `404` contract.

Coverage also allows false confidence. A test can execute both branches and contain no meaningful assertion. One quick audit is mutation: temporarily change `404` to `200` or reverse the condition. If the suite stays green, execution was measured but behavior was not protected.

Do not chase one universal percentage. A parser with many decisions may deserve dense branch coverage; a thin adapter may be better protected by a few real integration cases. Use thresholds to prevent accidental large regressions only after the baseline is understood. Exclude generated or unreachable platform code with a written reason so the report remains honest.

Run coverage, choose one untested error branch, and add a behavior-focused test for it. Mutate the branch afterward and confirm the new test fails for the expected public outcome.

Build a CI test pipeline

Order fast checks, integration dependencies, browser tests, artifacts, and cancellation so feedback is useful and affordable.

CI should reject broken changes quickly and retain enough evidence to diagnose slower failures.

Run formatting, types, and unit tests first. Cache dependencies safely, then run integration and browser jobs. Upload reports and traces only when useful, pin tool and service versions, and cancel superseded runs on the same branch.

Design the pipeline as evidence stages:

> **Diagram: CI test pipeline.** Static checks and unit tests run first, then a built artifact feeds integration and browser tests, followed by controlled performance smoke tests.

1. Static checks catch formatting, type, and import failures without starting services.
2. Unit tests catch deterministic domain regressions in seconds.
3. Integration tests start pinned databases and prove adapters, migrations, and HTTP contracts.
4. Browser tests prove a small set of critical journeys in the built application.
5. Performance smoke tests run only in a controlled environment with explicit thresholds.

Fast feedback does not require one serial job. Static and unit checks can run together if both are cheap. Slower jobs can start after their required artifact exists and run in parallel when they use isolated services. Make readiness explicit; "the process started" is weaker than "the health endpoint answers and the database accepts connections."

Give every job a timeout and a distinct failure story. A product assertion, dependency-install failure, unavailable container runtime, and server-readiness timeout need different evidence. Retain unit reports, service logs, and Playwright traces on failure. Do not upload large artifacts from every green run unless they serve a real audit need.

Pin runtime and service versions, but update them deliberately. Cache packages by the dependency manifest rather than a broad branch name. Cancel superseded runs for the same branch so old browser jobs do not consume capacity or publish stale results.

Write a pipeline outline for the Books app with commands, required services, readiness checks, timeouts, and retained artifacts. For every job, state whether a failure means the product is wrong or the test environment is unavailable, and what evidence lets you tell.

Add a k6 smoke test

Send a small controlled workload to the Books API and define thresholds that turn latency and error expectations into pass or fail.

Load testing asks how the system behaves under traffic. Begin with a tiny smoke run against infrastructure you own.

A k6 script describes virtual users, duration, requests, checks, and thresholds. Protect shared systems: start small, use test data, coordinate larger runs, and stop if error rate or resource pressure becomes unsafe.

```
import http from 'k6/http'
import { check } from 'k6'

export const options = {
  vus: 2,
  duration: '10s',
  thresholds: {
    checks: ['rate>0.99'],
    http_req_failed: ['rate<0.01'],
    http_req_duration: ['p(95)<300']
  }
}

export default function () {
  const response = http.get('http://localhost:3000/books')
  check(response, { 'status is 200': r => r.status === 200 })
}
```

A check records whether each response has the expected content. It does not, by itself, make k6 exit with a failed status. A threshold is the pass/fail contract for the run, so the `checks` threshold connects failed checks to CI. The other thresholds say that fewer than 1% of HTTP requests may fail and that 95% must complete within 300 milliseconds.

The percentile matters. Averages can hide a slow tail: nine fast requests and one very slow request may still produce an acceptable average while a real user experiences the outlier. Choose values from an actual service expectation or measured baseline, not from a number that merely makes today's run green.

A smoke test asks whether the system survives a tiny controlled workload and produces valid responses. It is not a capacity claim. Two virtual users for ten seconds cannot tell you the maximum throughput. Larger tests need representative traffic, isolated test data, observability, and coordination with whoever owns the environment.

Run against infrastructure you own and start locally or in a dedicated environment. A fast `200` error page is still wrong, so check both status and a small stable body property. Avoid including credentials or unbounded test-data creation in the script.

Run the smoke test locally and record request count, p95 latency, error rate, and check rate. Break the response body while keeping status `200`, then make the route slow. Confirm that the two changes fail different thresholds.

Finish the testing strategy

Create a maintainable suite with clear ownership, documented commands, deliberate gaps, and a process for fixing flaky tests.

A suite is a product that needs maintenance. Slow, duplicated, ownerless tests eventually become ignored tests.

Document each test layer, local and CI commands, required services, data isolation, and failure artifacts. When a test flakes, record and fix the cause or quarantine it visibly with an owner and deadline; do not silently retry forever.

Turn the course into a risk matrix rather than a folder inventory. Useful columns are risk, observable evidence, real boundary, test location, runtime, owner, and known gap. For example, "duplicate ISBN cannot overwrite a book" needs database constraint evidence; "reader can add a book" needs visible browser evidence. The matrix exposes duplicated tests that cross the same boundary and important risks that no test owns.

Map changes to feedback. A validation edit should get immediate unit results and the relevant HTTP contract tests. A migration edit must run clean-install, supported-upgrade, and repository tests. A shared UI change should run the critical browser journeys. The full suite still runs before release, but developers should not wait for every browser and performance test to learn that a pure function is wrong.

Define a flaky-test policy before flakiness becomes normal. Record the first failure's trace and environment, assign an owner, and fix shared state, timing, or infrastructure. If quarantine is necessary, keep it visible with a reason and expiry date. Retries may collect evidence, but an eventually green result does not erase the first failure.

Review the suite by asking what each test uniquely proves. Delete tests that duplicate stronger evidence without providing faster diagnosis. Keep deliberate gaps explicit: exhaustive browser validation may be too expensive when unit and HTTP tests already protect the rule.

Write the Books app test matrix from pure validation through API, database, browser, and k6 smoke coverage. Mark one intentional gap, one duplicated test to remove, and one failure artifact each layer must retain. Then give every flaky or quarantined test an owner and review date.

Check your understanding: reliability and ci

Check coverage use, CI stages, load test thresholds, flaky-test handling, suite ownership, and the final risk-based testing strategy.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/testing/.