Engineer

End-to-end test

a test that drives the whole system the way a user does

What it is

An end-to-end test checks a whole running system the way a user meets it. It starts the application with its real front end, back end, database and configuration, walks through a journey such as signing up or paying for an order, and asserts on what the user would see at the end. The usual shorthand is E2E test, and the practice is called end-to-end testing.

In *Software Engineering at Google*, Adam Bender sorts tests along two axes. Size is about resources: a small test stays inside one process; a medium test may use several processes on one machine; a large test may span machines. Scope is about how much code a test exercises, and tests of large scope go by several names, among them functional tests, end-to-end tests and system tests. Joseph Graves, in the book's chapter on larger testing, adds that the tests with the widest scope usually bring in several real dependencies and use fewer test doubles.

What those real dependencies buy is fidelity, which Graves defines as how well a test reflects the real behaviour of the system under test. A unit test of a checkout form proves that the form's code is right in a sandbox; an end-to-end test of the same form proves that the deployed page, the route behind it, the pricing rules and the database agree with one another. The cost is hermeticity: the more an environment resembles production, the harder it is to seal it off from everything else, and Graves notes that the two are often in direct conflict.

For a web application the user's interface is the browser, so that is where an end-to-end test stands. Graves puts it plainly: for end users, "the public API is the application itself". Playwright Test, which the example below uses, is the runner

the Playwright project builds for end-to-end tests of web applications. It bundles the runner, assertions, isolation and parallel runs, and drives Chromium, Firefox and WebKit, headless or headed.

Use the concept in a decision

Put “What is an end-to-end test?” to work in a real decision. We can help relate the concept to the system you are building or buying. One team taking an idea from a sketch to something your customers can log into, instead of handing it between three suppliers.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Led a GIS map application to 10x revenue growth — 10x revenue growth, led from the map product

Led a GIS map application to 10x revenue growth Get in touch →

Why it matters

Smaller tests leave gaps that only a test of the assembled system can close, and Graves lists the common ones. A test double can drift away from the thing it replaces, and nothing tells the test that the real implementation has changed. A unit test covers the code inside one binary, while the deployment configuration, start-up scripts and settings of the production instances sit outside it. Problems that appear only under load are a third gap, and the fourth is behaviour nobody anticipated, which is how most problems reach users in the first place. Bender writes that Google keeps large tests mostly for end-to-end runs of the whole system, which check its configuration more than any one piece of code.

They also check the assembled interface that the user meets. A form that posts to a route the back end has just renamed, or a release whose configuration points at the wrong service, passes every unit test on both sides and fails the first time a real person tries it.

The price is speed and reliability. Graves notes that by default Google gives a large test 15 minutes or an hour before it times out, and that they may be nonhermetic, sharing resources with other tests and traffic, and therefore nondeterministic. Bender names the usual sources of that nondeterminism: the clock, network delays, thread execution order, hardware interrupts and the browser’s renderer. That is why the book’s rough guideline is a pyramid: about 80 per cent narrow-scoped unit tests, 15 per cent integration tests and 5 per cent tests that exercise the complete system from end to end. Larger tests, Bender

writes, serve as a sanity check while the product grows and should not be the main way a team catches bugs, and Google teams often reserve them for builds and releases, keeping their cost out of the faster development loop.

What it looks like in practice

Take a shop that sells a teapot and asks a carrier's API for a delivery estimate. A Playwright spec for a guest's first purchase reads like the journey itself:

```
// e2e/checkout.spec.ts
import { test, expect } from "@playwright/test";

test.beforeEach(async ({ page }) => {
    // The carrier's API belongs to someone else: answer for it here.
    await page.route("https://api.carrier.example/**", (route) =>
        route.fulfill({ status: 200, body: '{"days": 2}' }),
    );
});

test("a guest can buy a teapot", async ({ page }) => {
    await page.goto("/shop/teapot");
    await page.getByRole("button", { name: "Add to basket" }).click();
    await page.getByRole("link", { name: "Basket" }).click();
    await expect(page.getByText("Delivery in 2 days")).toBeVisible();
    await page.getByLabel("Email").fill("buyer@example.com");
    await page.getByRole("button", { name: "Place order" }).click();
    const thanks = page.getByRole("heading", { name: "Thank you" });
    await expect(thanks).toBeVisible();
    await page.getByRole("link", { name: "View your order" }).click();
    await expect(page.getByText("Teapot")).toBeVisible();
});
```

Each locator names what the user sees: a button called Add to basket, a field labelled Email, a heading that says Thank you. Playwright recommends role locators first, and its locators are strict: when more than one element matches, the action throws instead of guessing. Before each click or fill, Playwright waits for the element to be actionable, and an assertion such as `toBeVisible()` keeps retrying until the condition holds, so the spec contains no sleeps. The `page.route` call answers the browser's requests to the carrier with a fixed body, so the test never depends on a server the team does not run; the glob has to match the whole URL, and `**` matches across slashes. Routing reaches only the requests the page itself makes: when the shop's server asks the carrier instead, the browser never sees that request, and a test double has to stand in on the server's

side. Every test also gets a fresh browser context, which Playwright likens to a brand-new browser profile, so the cookies and basket of one test cannot leak into the next.

The configuration starts the application and tells Playwright where it lives:

```
// playwright.config.ts
import { defineConfig } from "@playwright/test";

export default defineConfig({
    testDir: "e2e",
    retries: process.env.CI ? 2 : 0,
    use: {
        baseURL: "http://localhost:3000",
        trace: "on-first-retry",
    },
    webServer: {
        command: "npm run start",
        url: "http://localhost:3000",
        reuseExistingServer: !process.env.CI,
    },
});
```

The `webServer` block runs `npm run start` and waits until the URL answers before the first test; on a laptop it reuses a server that is already running, and on CI it always starts its own. `baseURL` lets the spec say `/shop/teapot` instead of a full address. `npx playwright test` runs the suite headless, `--headed` shows the browser window, and `npx playwright show-report` opens the HTML report. Playwright retries no failing test by default; here CI allows two retries, a test that fails and then passes is reported as flaky rather than passed, and a trace is recorded on the first retry to show what was different.

At a larger scale the browser layer is the smallest share of the work. The test suite built across Go, SQL and the browser, from 2025, has 981 Go test functions across 310 files, 543 Vitest and Playwright specs including 48 end-to-end files and 30 browser specs, and 494 SQL behavioural test files. The task was to check each layer's behaviour where that behaviour lives rather than everything from the outside through a browser, so a stored function is tested in the database instead of through three layers of application above it. The page is plain about the cost: the suite is slow, it taxes every change, and at that size it needs maintenance of its own.

Where it goes wrong

The suite is an ice cream cone. Bender's name for a suite heavy on end-to-end tests and light on integration and unit tests. A suite of that shape is usually slow, unreliable and awkward to work with, and the shape is common in projects that went from prototype to production in a hurry. Graves adds that when the code is hard to unit test, end-to-end tests are the only ones anyone can write, and the project has produced legacy code within days. Each new service also multiplies the paths through the system, so an end-to-end suite can never cover them all. Push each case down to the smallest test that can catch it, keep the browser for the journeys only the whole system can show, and build towards the pyramid from the first days of a project.

Selectors follow the markup. A locator such as `button.buttonIcon.episode-actions-later`, Playwright's own example of what to avoid, breaks the next time someone renames a class, although the user sees nothing different. Graves warns that UI tests have a reputation for being unreliable and expensive, partly because changes to look and feel break them while the behaviour stays the same. Locate elements by role, label or text, the way a user finds them, or by a test id the team owns. Where the back end has a public API, split the journey at the boundary between UI and API, and drive the rest of the end-to-end test through the API.

The test sleeps. A fixed wait is too short on a slow day and too long on every other day. Graves notes that tests relying on sleeps and timeouts all start failing when the machines that run them are overloaded, and the reruns add more load. Playwright's best practices show a subtler version of the same mistake: `expect(await page.getByText("welcome").isVisible()).toBe(true)` checks once and moves on, while `await expect(page.getByText("welcome")).toBeVisible()` waits for the condition. Use web-first assertions, and when a test waits on the back end, poll for the state change with a timeout.

Tests share data. One test creates an account that the next one expects, or two parallel workers buy the last teapot in stock. The suite passes in one order and fails in another, or only when it runs in parallel. Playwright's advice is to keep each test completely isolated, with its own storage, session, data and cookies, and to control the data in any database the tests use. Create what each test needs, with unique names, and never rely on what an earlier test left behind.

Someone else's server is in the test. A real payment provider, carrier or maps service makes the suite depend on a system the team cannot control, and sending it traffic may cost money. Graves advises against letting automated tests call a real third-party API, and calls such a dependency an important seam along which to split tests; Playwright's own advice is to test only what you control. Answer for the third party with `page.route` when the browser makes the call, as the spec above does, or with a test double the server is configured to call when the server makes it, and let that dependency be where the test ends.

Retries hide the flakiness. Retrying on CI turns a red build green, and the cost compounds. Bender works through the arithmetic: at 10,000 tests a day, a 0.1 per cent chance of a false failure in each means 10 flakes a day to investigate, and as flakiness approaches 1 per cent the tests begin to lose value. Rerunning trades machine time for engineering time and only postpones the root cause. The pytest documentation makes the same point from another angle: tests at a higher level depend on more state, and so fail intermittently more often. Treat every test Playwright reports as flaky as a bug, open the trace from the retry, and fix the test or delete it.

Further reading

- Testing Overview — Adam Bender's chapter of *Software Engineering at Google*: test size and scope, the pyramid, the ice cream cone and the cost of flaky tests.
- Larger Testing — Joseph Graves on fidelity, the gaps unit tests leave, UI and third-party boundaries, and driving out flakiness.
- Installation — Playwright's own introduction: installing Playwright Test, running the example test, headed mode and the HTML report.
- Best Practices — testing user-visible behaviour, isolating tests, leaving third-party servers out, locators and web-first assertions.
- Retries — how Playwright retries failing tests and labels each one passed, flaky or failed.
- Flaky tests — pytest's explanation of why tests flake, from uncontrolled system state and test order to thread safety, and what to do about it.



Glossary

Choose for your actual constraints

Describe the workflow, the existing interfaces and the users who need the change. We can define a testable delivery and a handover that lets your team maintain it.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is an end-to-end test?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/end-to-end-test/