Engineer

End-to-End-Test

ein Test, der das ganze System wie ein Nutzer bedient

Was es ist

Ein End-to-End-Test prüft ein ganzes laufendes System so, wie ein Nutzer es erlebt. Er startet die Anwendung mit echtem Frontend, Backend, Datenbank und Konfiguration, durchläuft einen Weg wie Registrierung oder Bezahlung einer Bestellung und prüft, was der Nutzer am Ende sehen würde. Die übliche Kurzform ist E2E-Test, die Praxis heißt End-to-End-Testing.

In *Software Engineering at Google* ordnet Adam Bender Tests auf zwei Achsen ein. Größe betrifft Ressourcen: Ein kleiner Test bleibt in einem Prozess, ein mittlerer kann mehrere Prozesse auf einem Rechner verwenden, ein großer mehrere Rechner umfassen. Umfang beschreibt, wie viel Code ein Test ausführt. Tests mit großem Umfang heißen unter anderem funktionale Tests, End-to-End-Tests und Systemtests. Joseph Graves ergänzt im Buchkapitel über größere Tests, dass Tests mit dem größten Umfang meist mehrere echte Abhängigkeiten und weniger Testdoubles einbeziehen.

Diese echten Abhängigkeiten liefern Realitätsnähe. Graves definiert sie danach, wie gut ein Test das echte Verhalten des geprüften Systems abbildet. Ein Unit-Test eines Checkoutformulars belegt die Korrektheit seines Codes in einer Sandbox. Ein End-to-End-Test desselben Formulars belegt, dass bereitgestellte Seite, dahinterliegende Route, Preisregeln und Datenbank übereinstimmen. Der Preis ist eine schlechtere Abschottung: Je näher eine Umgebung an Produktion liegt, desto schwerer lässt sie sich von allem anderen isolieren. Laut Graves stehen beide Eigenschaften oft in direktem Konflikt.

Bei einer Webanwendung ist der Browser die Nutzerschnittstelle, dort setzt deshalb der End-to-End-Test an. Graves sagt es klar: Für Endnutzer ist „die öffentliche API die Anwendung selbst“. Playwright Test, der Runner des folgenden Beispiels, wird vom Playwright-Projekt für End-to-End-Tests von

Webanwendungen gebaut. Er bündelt Runner, Zusicherungen, Isolation und parallele Läufe und steuert Chromium, Firefox und WebKit mit oder ohne sichtbares Fenster.

Das Konzept für eine Entscheidung nutzen

Nutzen Sie „Was ist ein End-to-End-Test?“ für eine konkrete Entscheidung. Wir können helfen, das Konzept mit dem System in Verbindung zu bringen, das Sie entwickeln oder einkaufen. Ein Team führt eine Idee von der Skizze bis zu einem Produkt, bei dem Ihre Kunden sich anmelden können, statt sie zwischen drei Dienstleistern weiterzureichen.

Unsere Leistungen entdecken Kontakt aufnehmen →

Die Annahmen hinter dem Begriff prüfen

Nutzen Sie die Erklärung und die verlinkten Arbeiten, um Fragen für Ihr Projekt zu formulieren. Wir können die Annahmen gemeinsam mit Ihnen prüfen, bevor ein Werkzeug ausgewählt wird.
GIS-Kartenprodukt zu 10-fachem Umsatz geführt — 10x Umsatzwachstum durch die Leitung des Kartenprodukts

GIS-Kartenprodukt zu 10-fachem Umsatz geführt Kontakt aufnehmen →

Warum es wichtig ist

Kleinere Tests lassen Lücken, die nur ein Test des zusammengesetzten Systems schließen kann. Graves nennt die üblichen: Ein Testdouble kann sich vom ersetzten Original entfernen, ohne dass der Test eine geänderte echte Implementierung bemerkt. Ein Unit-Test deckt den Code einer Binärdatei ab, nicht aber Bereitstellungskonfiguration, Startskripte und Einstellungen der Produktionsinstanzen. Nur unter Last auftretende Probleme sind die dritte Lücke. Die vierte ist unerwartetes Verhalten, über das die meisten Probleme überhaupt erst Nutzer erreichen. Laut Bender setzt Google große Tests vor allem für End-to-End-Läufe des Gesamtsystems ein. Diese prüfen dessen Konfiguration mehr als einzelne Codeteile.

Sie prüfen außerdem die Oberfläche des zusammengesetzten Systems, mit der der Nutzer arbeitet. Ein Formular, das an eine vom Backend gerade umbenannte Route sendet, oder ein Release, dessen Konfiguration auf den falschen Dienst verweist, besteht auf beiden Seiten jeden Unit-Test und scheitert beim ersten echten Nutzer.

Das kostet Geschwindigkeit und Zuverlässigkeit. Laut Graves gibt Google einem großen Test standardmäßig 15 Minuten oder eine Stunde bis zur Zeitüberschreitung. Solche Tests sind unter Umständen nicht abgeschottet, teilen Ressourcen mit anderen Tests und Verkehr und sind damit nicht deterministisch. Bender nennt die üblichen Ursachen: Uhrzeit, Netzwerkverzögerungen, Reihenfolge der Threadausführung, Hardwareinterrupts und den Browserrenderer. Deshalb empfiehlt das Buch grob

eine Pyramide: etwa 80 Prozent eng begrenzte Unit-Tests, 15 Prozent Integrationstests und 5 Prozent Tests des ganzen Systems von Ende zu Ende. Größere Tests dienen laut Bender beim Produktwachstum als Plausibilitätsprüfung und sollten nicht der Hauptweg zur Fehlererkennung sein. Google-Teams reservieren sie oft für Builds und Releases, damit ihre Kosten die schnellere Entwicklungsschleife nicht belasten.

Wie es in der Praxis aussieht

Ein Shop verkauft eine Teekanne und fragt die API eines Versanddienstleisters nach einer Lieferprognose. Eine Playwright-Spezifikation für den ersten Gastkauf liest sich wie der Ablauf selbst:

```
// 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();
});
```

Jeder Locator benennt, was der Nutzer sieht: einen Button namens Add to basket, ein mit Email beschriftetes Feld, eine Überschrift Thank you. Playwright empfiehlt zuerst rollenbasierte Locators. Seine Locators sind strikt: Passt mehr als ein Element, löst die Aktion eine Ausnahme aus, statt zu raten. Vor jedem Klick oder Ausfüllen wartet Playwright auf die Bedienbarkeit des Elements. Eine Zusicherung wie `toBeVisible()` versucht es bis zum Eintreten der Bedingung erneut. Die Spezifikation enthält daher keine festen Wartepausen. Der Aufruf `page.route` beantwortet die Browseranfragen an den Versanddienstleister mit

einem festen Body. So hängt der Test nie von einem fremdbetriebenen Server ab. Das Globmuster muss auf die ganze URL passen; `**` erfasst auch Schrägstriche. Das Routing erreicht nur Anfragen, die die Seite selbst stellt. Fragt stattdessen der Shopserver den Versanddienstleister, sieht der Browser diese Anfrage nie. Ein Testdouble muss dann auf der Serverseite einspringen. Jeder Test erhält auch einen frischen Browserkontext, den Playwright mit einem ganz neuen Browserprofil vergleicht. Cookies und Warenkorb eines Tests können so nicht in den nächsten gelangen.

Die Konfiguration startet die Anwendung und sagt Playwright, wo sie erreichbar ist:

```
// 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,
    },
});
```

Der Block `webServer` führt `npm run start` aus und wartet vor dem ersten Test auf eine Antwort der URL. Auf einem Laptop nutzt er einen schon laufenden Server weiter, in der CI startet er immer einen eigenen. Mit `baseURL` kann die Spezifikation `/shop/teapot` statt einer vollständigen Adresse angeben. `npx playwright test` führt die Testsuite ohne Fenster aus, `--headed` zeigt das Browserfenster und `npx playwright show-report` öffnet den HTML-Bericht. Standardmäßig wiederholt Playwright keinen fehlgeschlagenen Test. Hier erlaubt CI zwei Wiederholungen. Ein erst fehlgeschlagener und dann bestandener Test wird als flaky statt als bestanden gemeldet. Beim ersten Wiederholungsversuch wird ein Trace aufgezeichnet, der die Unterschiede zeigt.

In größerem Maßstab ist die Browserschicht der kleinste Arbeitsanteil. Die über Go, SQL und Browser aufgebaute Testsuite aus der seit 2025 laufenden Tätigkeit umfasst 981 Go-Testfunktionen in 310 Dateien, 543 Vitest- und Playwright-Spezifikationen einschließlich 48 End-to-End-Dateien und 30 Browserspezifikationen sowie 494 SQL-Dateien mit Verhaltenstests. Die Aufgabe war, das Verhalten jeder Schicht dort zu prüfen, wo es liegt, statt alles von außen durch einen Browser. Eine gespeicherte Funktion wird deshalb in der Datenbank und nicht durch drei darüberliegende Anwendungsschichten getestet. Die Seite benennt die Kosten klar: Die Testsuite ist langsam, belastet jede Änderung und braucht in dieser Größe eigene Pflege.

Wo es schiefgeht

Die Testsuite ist eine Eistüte. So nennt Bender eine Suite mit vielen End-to-End-Tests und wenigen Integrations- und Unit-Tests. Sie ist meist langsam, unzuverlässig und umständlich in der Arbeit. Diese Form ist bei eilig vom Prototyp in Produktion gebrachten Projekten verbreitet. Graves ergänzt: Lässt sich der Code schwer mit Unit-Tests prüfen, sind End-to-End-Tests die einzigen schreibbaren Tests, und das Projekt hat binnen Tagen Altcode produziert. Jeder neue Dienst vervielfacht zudem die Systemwege; eine End-to-End-Suite kann nie alle abdecken. Verschieben Sie jeden Fall in den kleinsten Test, der ihn erkennt. Behalten Sie den Browser für die Abläufe, die nur das Gesamtsystem zeigen kann, und arbeiten Sie von den ersten Projekttagen an auf die Pyramide hin.

Selektoren folgen dem Markup. Ein Locator wie `button.buttonIcon.episode-actions-later`, Playwrights eigenes Negativbeispiel, scheitert bei der nächsten Klassenumbenennung, obwohl der Nutzer keinen Unterschied sieht. Graves warnt: UI-Tests gelten als unzuverlässig und teuer, unter anderem weil Änderungen am Erscheinungsbild sie bei gleichem Verhalten brechen. Finden Sie Elemente anhand von Rolle, Beschriftung oder Text, wie ein Nutzer, oder anhand einer vom Team kontrollierten Test-ID. Hat das Backend eine öffentliche API, teilen Sie den Ablauf an der Grenze zwischen UI und API und steuern Sie den restlichen End-to-End-Test über die API.

Der Test schläft. Eine feste Wartezeit ist an langsamen Tagen zu kurz und sonst zu lang. Laut Graves beginnen von Wartepausen und Zeitüberschreitungen abhängige Tests bei überlasteten Testrechnern alle zu scheitern. Ihre Wiederholungen erhöhen die Last weiter. Playwrights bewährte Praktiken zeigen eine subtilere Form desselben Fehlers: `expect(await page.getByText("welcome").isVisible()).toBe(true)` prüft einmal und macht weiter, während `await expect(page.getByText("welcome")).toBeVisible()` auf die Bedingung

wartet. Nutzen Sie Zusicherungen, die auf Webzustände warten. Wartet ein Test auf das Backend, fragen Sie den Zustandswechsel wiederholt mit einer Zeitgrenze ab.

Tests teilen Daten. Ein Test erstellt ein vom nächsten erwartetes Konto, oder zwei parallele Worker kaufen die letzte Teekanne im Lager. Die Suite besteht in einer Reihenfolge und scheitert in einer anderen oder nur bei paralleler Ausführung. Playwright empfiehlt die vollständige Isolation jedes Tests mit eigenem Speicher, eigener Sitzung, eigenen Daten und Cookies sowie die Kontrolle über die Daten jeder verwendeten Datenbank. Erstellen Sie mit eindeutigen Namen, was jeder Test braucht, und verlassen Sie sich nie auf Hinterlassenschaften früherer Tests.

Ein fremder Server steckt im Test. Ein echter Zahlungsanbieter, Versanddienstleister oder Kartendienst macht die Suite von einem vom Team nicht kontrollierbaren System abhängig. Sein Aufruf kann Geld kosten. Graves rät von echten Drittanbieter-API-Aufrufen durch automatisierte Tests ab und nennt solche Abhängigkeiten eine wichtige Trennlinie für Tests. Playwright empfiehlt selbst, nur Kontrollierbares zu prüfen. Antworten Sie für den Drittanbieter per `page.route`, wenn der Browser anfragt, wie oben, oder konfigurieren Sie den Server für den Aufruf eines Testdoubles, wenn er die Anfrage stellt. Lassen Sie den Test an dieser Abhängigkeit enden.

Wiederholungen verbergen instabile Tests. Wiederholungen in der CI machen einen roten Build grün, und die Kosten summieren sich. Bender rechnet vor: Bei 10.000 Tests täglich ergeben 0,1 Prozent Wahrscheinlichkeit eines falschen Fehlschlags pro Test 10 zu untersuchende instabile Läufe am Tag. Nähert sich die Instabilität 1 Prozent, verlieren die Tests an Wert. Wiederholungen ersetzen Entwicklungszeit durch zusätzliche Rechnerzeit und schieben die Behebung der Ursache nur auf. Die pytest-Dokumentation zeigt denselben Punkt aus einem anderen Blickwinkel: Tests auf höherer Ebene hängen von mehr Zustand ab und scheitern deshalb häufiger sporadisch. Behandeln Sie jeden von Playwright als flaky gemeldeten Test als Bug, öffnen Sie den Trace der Wiederholung und reparieren oder löschen Sie den Test.

Weiterführende Lektüre

- Testüberblick — Adam Benders Kapitel aus *Software Engineering at Google*: Testgröße und -umfang, Pyramide, Eistüte und die Kosten instabiler Tests.

- Größere Tests — Joseph Graves über Realitätsnähe, Lücken von Unit-Tests, Grenzen zu UI und Drittanbietern sowie das Beseitigen instabiler Tests.
- Installation — Playwrights Einführung: Playwright Test installieren, den Beispieltest ausführen, Modus mit Fenster und HTML-Bericht.
- Bewährte Praktiken — nutzersichtbares Verhalten prüfen, Tests isolieren, Drittanbieterserver weglassen, Locators und Zusicherungen, die auf Webzustände warten.
- Wiederholungen — wie Playwright fehlgeschlagene Tests wiederholt und jeweils als bestanden, flaky oder fehlgeschlagen kennzeichnet.
- Instabile Tests — pytest erklärt Instabilität durch unkontrollierten Systemzustand, Testreihenfolge und Threadsicherheit sowie Gegenmaßnahmen.

Glossar

Für Ihre tatsächlichen Rahmenbedingungen auswählen

Beschreiben Sie den Arbeitsablauf, die vorhandenen Schnittstellen und die Nutzer, die die Veränderung benötigen. Wir können eine prüfbare Lieferung definieren und die Übergabe so gestalten, dass Ihr Team die gelieferten Bestandteile warten kann.

Preise ansehen Kontakt aufnehmen →

Die Begriffe praktisch nutzbar machen

Fragen Sie, wie „Was ist ein End-to-End-Test?“ auf Ihre Organisation zutrifft. Ein konkretes Beispiel aus Ihrem bestehenden System reicht aus, um das Gespräch zu beginnen.

Unsere Leistungen entdecken

welcome@engineer.company +45 50 10 61 84 Kopenhagen, Dänemark

Nehmen Sie diese Seite mit

Kontaktkarte

Mit Sorgfalt und Liebe gestaltet von D.

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

Copyright © 2025 – heute · Engineer ApS · Alle Rechte vorbehalten.

Gedruckt von https://engineer.company/de/glossary/end-to-end-test/