Engineer

Test de bout en bout

🏃 test qui parcourt tout le système comme un utilisateur

Ce que c’est

Un test de bout en bout vérifie un système complet en fonctionnement, tel que l’utilisateur le rencontre. Il démarre l’application avec ses composants réels : interface, partie serveur, base de données et configuration. Il suit un parcours comme l’inscription ou le paiement d’une commande, puis vérifie ce que l’utilisateur verrait à la fin. L’abréviation habituelle est test E2E ; la pratique s’appelle le test de bout en bout.

Dans *Software Engineering at Google*, Adam Bender classe les tests selon deux axes. La taille concerne les ressources : un petit test reste dans un processus ; un test moyen peut employer plusieurs processus sur une machine ; un grand test peut s’étendre sur plusieurs machines. La portée concerne la quantité de code exercée. Les tests de grande portée portent plusieurs noms, dont tests fonctionnels, tests de bout en bout et tests système. Dans le chapitre du livre consacré aux tests de plus grande taille, Joseph Graves ajoute que les tests à la portée la plus large intègrent généralement plusieurs dépendances réelles et utilisent moins de doublures de test.

Ces dépendances réelles apportent de la fidélité, que Graves définit comme la mesure dans laquelle un test reflète le comportement réel du système testé. Un test unitaire de formulaire de paiement prouve que le code du formulaire est correct dans un environnement isolé ; un test de bout en bout du même formulaire prouve que la page déployée, la route derrière elle, les règles de tarification et la base de données s’accordent. Le coût est l’herméticité : plus un environnement ressemble à la production, plus il est difficile de l’isoler de tout le reste. Graves note que les deux sont souvent en conflit direct.

Pour une application web, l'utilisateur passe par le navigateur ; c'est donc là que se place le test de bout en bout. Graves le dit simplement : pour les utilisateurs finaux, « l'API publique est l'application elle-même ». Playwright Test, utilisé dans l'exemple ci-dessous, est l'outil d'exécution que le projet Playwright construit pour les tests de bout en bout d'applications web. Il rassemble l'exécution, les assertions, l'isolation et les exécutions parallèles, et pilote Chromium, Firefox et WebKit, avec ou sans fenêtre visible.

Utilisez le concept pour prendre une décision

Utilisez « Qu'est-ce qu'un test de bout en bout ? » pour éclairer une décision réelle. Nous pouvons vous aider à relier ce concept au système que vous construisez ou achetez. Une seule équipe transforme une idée en un produit auquel vos clients peuvent se connecter, sans le faire passer entre trois prestataires.

Découvrir nos prestations Contactez-nous →

Vérifiez les hypothèses derrière le terme

Appuyez-vous sur l'explication et les réalisations liées pour formuler les questions de votre projet. Nous pouvons examiner les hypothèses avec vous avant le choix d'un outil. Application GIS : revenus multipliés par 10 — 10x croissance du chiffre d'affaires portée par le produit cartographique

Application GIS : revenus multipliés par 10 Contactez-nous →

Pourquoi c'est important

Les tests plus petits laissent des lacunes que seul un test du système assemblé peut combler. Graves en dresse la liste habituelle. Une doublure de test peut s'éloigner de ce qu'elle remplace, sans que rien ne prévienne le test que l'implémentation réelle a changé. Un test unitaire couvre le code d'un binaire ; la configuration de déploiement, les scripts de démarrage et les paramètres des instances de production restent en dehors. Les problèmes qui n'apparaissent qu'en charge constituent une troisième lacune. La quatrième est le comportement que personne n'avait anticipé, par lequel la plupart des problèmes atteignent les utilisateurs. Bender écrit que Google réserve principalement les grands tests aux parcours de bout en bout du système complet, qui vérifient davantage sa configuration qu'un morceau de code particulier.

Ils vérifient aussi l'interface du système assemblé que rencontre l'utilisateur. Un formulaire qui envoie ses données à une route que le backend vient de renommer, ou une version dont la configuration pointe vers le mauvais service, passe tous les tests unitaires des deux côtés et échoue dès qu'une personne réelle l'essaie.

La contrepartie est une perte de vitesse et de fiabilité. Graves note que, dans sa configuration standard, Google accorde à un grand test 15 minutes ou une heure avant son expiration. Ces tests peuvent aussi être non hermétiques, partager des ressources avec d'autres tests et du trafic, et donc ne pas être déterministes. Bender nomme les sources habituelles de ce non-déterminisme : l'horloge, les délais réseau, l'ordonnancement des fils d'exécution, les interruptions matérielles et le moteur de rendu du navigateur. Le livre propose donc une pyramide comme règle approximative : environ 80 % de tests unitaires de portée étroite, 15 % de tests d'intégration et 5 % de tests exerçant tout le système de bout en bout. Selon Bender, les tests de plus grande taille servent de vérification de cohérence pendant la croissance du produit ; ils ne devraient pas être le principal moyen de détecter les bugs. Les équipes de Google les réservent souvent aux étapes de construction et aux mises à disposition, pour en écarter le coût de la boucle de développement plus rapide.

À quoi cela ressemble en pratique

Prenez une boutique qui vend une théière et demande une estimation de livraison à l'API d'un transporteur. Une spécification Playwright du premier achat d'un visiteur non connecté se lit comme le parcours lui-même :

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

Chaque localisateur nomme ce que l'utilisateur voit : un bouton nommé Add to basket, un champ libellé Email, un titre affichant Thank you. Playwright recommande d'abord les localisateurs par rôle, et ses localisateurs sont stricts : si plusieurs éléments correspondent, l'action lève une exception plutôt que de deviner. Avant chaque clic ou saisie, Playwright attend que l'élément soit utilisable. Une assertion telle que `toBeVisible()` réessaie jusqu'à ce que la condition soit satisfaite ; la spécification ne contient donc aucune attente fixe. L'appel à `page.route` répond aux requêtes du navigateur vers le transporteur avec un corps de réponse fixe : le test ne dépend jamais d'un serveur que l'équipe n'exploite pas. Le motif glob doit correspondre à l'URL entière, et `**` peut couvrir les barres obliques. Ce routage ne touche que les requêtes effectuées par la page elle-même. Si c'est le serveur de la boutique qui interroge le transporteur, le navigateur ne voit jamais cette requête ; une doublure de test doit alors intervenir côté serveur. Chaque test reçoit aussi un nouveau contexte de navigateur, que Playwright compare à un profil de navigateur neuf : les cookies et le panier d'un test ne peuvent donc pas se retrouver dans le suivant.

La configuration démarre l'application et indique à Playwright où elle se trouve :

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

Le bloc `webServer` lance `npm run start` et attend que l'URL réponde avant le premier test. Sur un portable, il réutilise un serveur déjà en fonctionnement ; en CI, il démarre toujours le sien. `baseURL` permet à la spécification d'indiquer `/shop/teapot` plutôt qu'une adresse complète. `npx playwright test` exécute la suite sans fenêtre visible, `--headed` affiche celle du navigateur et `npx playwright show-report` ouvre le rapport HTML. Par défaut, Playwright ne réessaie aucun test

en échec. Ici, la CI autorise deux nouvelles tentatives ; un test qui échoue puis réussit est signalé comme instable, plutôt que réussi. Une trace est enregistrée à la première nouvelle tentative pour montrer ce qui était différent.

À plus grande échelle, la couche navigateur est la plus petite part du travail. La suite de tests construite sur Go, SQL et le navigateur, construite dans le cadre d'un poste occupé depuis 2025, compte 981 fonctions de test Go réparties sur 310 fichiers, 543 spécifications Vitest et Playwright, dont 48 fichiers de bout en bout et 30 spécifications navigateur, ainsi que 494 fichiers de tests comportementaux SQL. Il s'agissait de vérifier le comportement de chaque couche là où il se produit, plutôt que de tout tester de l'extérieur au moyen d'un navigateur. Une fonction stockée est donc testée dans la base, plutôt qu'à travers trois couches d'application au-dessus d'elle. La page expose clairement le coût : la suite est lente, elle pèse sur chaque modification et, à cette taille, elle a besoin de sa propre maintenance.

Où les problèmes surviennent

La suite est un cornet de glace. C'est le nom que Bender donne à une suite riche en tests de bout en bout et pauvre en tests d'intégration et unitaires. Une telle suite est généralement lente, peu fiable et difficile à utiliser. Cette forme est courante dans les projets passés rapidement du prototype à la production. Graves ajoute que, lorsque le code se prête mal aux tests unitaires, les tests de bout en bout sont les seuls que l'on puisse écrire : le projet a produit du code hérité en quelques jours. Chaque nouveau service multiplie aussi les parcours à travers le système ; une suite de bout en bout ne pourra donc jamais tous les couvrir. Ramenez chaque cas au plus petit test capable de le détecter, réservez le navigateur aux parcours que seul le système entier peut montrer et construisez la pyramide dès les premiers jours du projet.

Les sélecteurs suivent le balisage. Un localisateur comme `button.buttonIcon.episode-actions-later`, l'exemple à éviter donné par Playwright, casse au prochain renommage de classe alors que l'utilisateur ne voit aucune différence. Graves rappelle que les tests d'interface ont la réputation d'être peu fiables et coûteux, en partie parce que les changements d'apparence les cassent sans modifier le comportement. Localisez les éléments par rôle, libellé ou texte, comme le ferait un utilisateur, ou par un identifiant de test maîtrisé par l'équipe. Lorsque le backend possède une API publique, coupez le parcours à la frontière entre interface et API, puis pilotez le reste du test de bout en bout par l'API.

Le test dort. Une attente fixe est trop courte les jours lents et trop longue tous les autres. Graves note que les tests reposant sur des pauses et des délais d'expiration commencent tous à échouer lorsque leurs machines sont surchargées ; les nouvelles exécutions augmentent encore la charge. Les bonnes pratiques de Playwright montrent une version plus subtile de la même erreur : `expect(await page.getByText("welcome").isVisible()).toBe(true)` ne vérifie qu'une fois avant de poursuivre, tandis que `await expect(page.getByText("welcome")).toBeVisible()` attend la condition. Employez les assertions conçues pour le web et, lorsqu'un test attend le backend, interrogez périodiquement l'état jusqu'au changement, avec un délai maximal.

Les tests partagent des données. Un test crée un compte que le suivant s'attend à trouver, ou deux processus parallèles achètent la dernière théière en stock. La suite réussit dans un ordre et échoue dans un autre, ou seulement en parallèle. Playwright conseille de garder chaque test complètement isolé, avec son propre stockage, sa session, ses données et ses cookies, et de contrôler les données de toute base utilisée par les tests. Créez ce dont chaque test a besoin, avec des noms uniques, sans jamais compter sur ce qu'un test précédent a laissé.

Le serveur d'un tiers fait partie du test. Un véritable prestataire de paiement, transporteur ou service cartographique rend la suite dépendante d'un système que l'équipe ne contrôle pas ; lui envoyer du trafic peut aussi coûter de l'argent. Graves déconseille de laisser les tests automatisés appeler une véritable API tierce et voit dans cette dépendance une frontière importante où séparer les tests. Playwright conseille lui aussi de ne tester que ce que l'on contrôle. Répondez à la place du tiers avec `page.route` si le navigateur effectue l'appel, comme dans la spécification ci-dessus, ou avec une doublure que le serveur est configuré pour appeler si c'est lui qui émet la requête. Faites de cette dépendance la limite du test.

Les nouvelles tentatives masquent l'instabilité. Réessayer en CI fait passer le résultat de la construction du rouge au vert, et le coût s'accumule. Bender détaille le calcul : sur 10 000 tests par jour, une probabilité de faux échec de 0,1 % pour chacun donne 10 échecs intermittents à examiner chaque jour. Lorsque l'instabilité approche 1 %, les tests commencent à perdre leur valeur. Réexécuter échange du temps machine contre du temps d'ingénierie et ne fait que repousser le traitement de la cause profonde. La documentation de pytest formule le même constat sous un autre angle : les tests de plus haut niveau dépendent de davantage d'éléments d'état et échouent donc plus souvent de

manière intermittente. Traitez chaque test signalé comme instable par Playwright comme un bug, ouvrez la trace de la nouvelle tentative et corrigez ou supprimez le test.

Pour aller plus loin

- Vue d'ensemble des tests — le chapitre d'Adam Bender dans *Software Engineering at Google* : taille et portée des tests, pyramide, cornet de glace et coût des tests instables.
- Tests de plus grande taille — Joseph Graves sur la fidélité, les lacunes des tests unitaires, les frontières avec l'interface et les tiers, et l'élimination de l'instabilité.
- Installation — l'introduction de Playwright : installation de Playwright Test, exécution de l'exemple, mode avec fenêtre visible et rapport HTML.
- Bonnes pratiques — tester le comportement visible, isoler les tests, exclure les serveurs tiers, employer les localisateurs et les assertions conçues pour le web.
- Nouvelles tentatives — comment Playwright réessaie les tests en échec et les classe en réussis, instables ou échoués.
- Tests instables — l'explication de pytest : état du système non maîtrisé, ordre des tests, sûreté en exécution multithread et remèdes.

Glossaire

Choisissez selon vos contraintes réelles

Présentez le processus de travail, les interfaces existantes et les utilisateurs qui ont besoin du changement. Nous pouvons définir une livraison testable et une transmission permettant à votre équipe d'assurer la maintenance des éléments livrés.

Consulter les tarifs Contactez-nous →

Rendez la terminologie concrète

Demandez comment « Qu'est-ce qu'un test de bout en bout? » s'applique à votre organisation. Un exemple concret tiré de votre système actuel suffit pour engager la discussion.

Découvrir nos prestations

welcome@engineer.company +45 50 10 61 84 Copenhague, Danemark

Emportez cette page avec vous

Fiche de contact

Créé avec soin et amour par D.

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

Droit d'auteur © 2025 – aujourd'hui · Engineer ApS · Tous droits réservés.

Imprimé depuis https://engineer.company/fr/glossary/end-to-end-test/