Engineer

エンドツーエンドテスト

🏃 利用者と同じようにシステム全体を操作するテスト

どのようなものか

エンドツーエンドテストは、動作中のシステム全体を、利用者が使う形で確かめます。本物のフロントエンド、バックエンド、データベース、設定でアプリを起動し、登録や注文の支払いといった一連の操作を進め、最後に利用者へ見える結果を検証します。通常はE2Eテストと略し、このように試すことを、英語ではend-to-end testingと呼びます。

_Software Engineering at Google_で、Adam Benderは、二つの軸でテストを分類します。サイズは、使う資源のことです。小さいテストは一つのプロセス内に収まり、中くらいのテストは一台で複数のプロセスを使うことがあり、大きいテストは複数のマシンにまたがることがあります。範囲は、どれだけのコードを動かすかです。範囲の広いテストには、機能テスト、エンドツーエンドテスト、システムテストなど、さまざまな名前があります。同書の大きいテストの章で、Joseph Gravesは、最も範囲の広いテストでは、複数の本物の依存先を組み込み、テストダブルをあまり使わないことが多いと補足します。

本物の依存先から得られるのは、再現の忠実度です。Gravesは、テスト対象システムの実際の振る舞いを、どれだけよく反映するかと定義します。購入フォームの単体テストは、隔離した環境で、そのフォームのコードが正しいことを示します。同じフォームのエンドツーエンドテストは、配備したページ、その背後のルート、価格のルール、データベースが、互いに整合していることを示します。代償として、外部環境からの隔離性が下がります。環境を本番へ近づけるほど、ほかのすべてから切り離すのは難しくなります。Gravesは、両者が直接衝突することが多いと指摘します。

ウェブアプリでは、利用者のインターフェースはブラウザーなので、エンドツーエンドテストも、そこを起点にします。Gravesの表現は明確です。エンドユーザーにとって「公開APIは、アプリケーションそのもの」です。次の例で使うPlaywright Testは、Playwrightプロジェクトが開発する、ウェブアプリのエンド

ツーエンドテスト用ランナーです。ランナー、アサーション、隔離、並列実行を備え、ウィンドウの表示の有無にかかわらず、Chromium、Firefox、WebKitを操作します。

概念を判断に生かす

「エンドツーエンドテストとは？」を実際の判断に生かしましょう。その概念を、開発または購入するシステムに結び付けて考えるお手伝いをします。 3社の間で受け渡す代わりに、1つのチームがアイデアのスケッチから、お客様がログインして使える製品まで担当します。

サービスを見る お問い合わせ →

用語の背後にある前提を検証する

解説とリンク先の実績を参考に、プロジェクトについての質問を整理してください。ツールを選ぶ前に、前提を一緒に検討できます。 GIS地図アプリの開発を主導し売上を10倍に — 10x 地図製品の開発を主導して実現した売上成長

GIS地図アプリの開発を主導し売上を10倍に お問い合わせ →

なぜ重要なのか

小さいテストには、組み上がったシステムのテストでしか埋められない穴が残ります。Gravesは、よくあるものを挙げます。テストダブルの動作が、置き換える対象からずれても、本物の実装が変わったことを、テストへ知らせるものはありません。単体テストが対象にするのは、一つのバイナリー内のコードです。デプロイの設定、起動スクリプト、本番インスタンスの設定は、その外にあります。三つ目は、負荷がかかったときだけ現れる問題です。四つ目は、誰も予想しなかった振る舞いで、そもそも多くの問題は、この形で利用者へ届きます。Benderによると、Googleは、大きいテストを主にシステム全体のエンドツーエンド実行に使います。個々のコードより、システムの構成を確かめるものです。

利用者が実際に操作する、組み上がったインターフェースも検査します。バックエンドが名前を変えたばかりのルートへ送信するフォームや、設定が違うサービスを指すリリースは、両側のすべての単体テストに通っても、実際の利用者が初めて使うと失敗します。

代償は、速度と信頼性です。Gravesによると、Googleでは、大きいテストのタイムアウトまでの時間を、既定で15分または一時間にします。ほかのテストやトラフィックと資源を共有し、外部から隔離されておらず、結果が一定にならないこともあります。Benderは、その原因として、時計、ネットワークの遅延、スレッドの実行順序、ハードウェア割り込み、ブラウザーの描画処理を挙げます。同書のおおまかな目安がピラミッドになるのは、そのためです。範囲の狭い単体テストが約80パーセント、結合テストが15パーセント、システム全体を端から端まで動かすテストが5パーセントです。Benderは、大きいテストを、製品の成長中に基本が崩れていないか確かめるものと位置付け、チームがバグを見つける

主な手段にすべきではないと述べます。Googleのチームは、ビルドやリリースのときに使うことが多く、速い開発サイクルへ、その実行の負担を持ち込みません。

実際の活用例

ティーポットを売り、配送日数の見積もりを配送業者のAPIへ問い合わせる店を考えます。会員登録をしない客の初回購入を試すPlaywrightのテスト仕様は、購入の手順そのもののように読めます。

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

test.beforeEach(async ({ page }) => {
    // 配送業者のAPIは外部にあるため、ここで代わりに応答する
    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();
});
```

各ロケーターは、利用者が見るものを名指しします。「Add to basket」というボタン、「Email」というラベルの入力欄、「Thank you」と書かれた見出しです。Playwrightは、まずロールによるロケーターを勧めます。ロケーターは厳密で、複数の要素が一致すると、推測せず、操作が例外を投げます。クリックや入力の前には、要素を操作できるまで待ち、`toBeVisible()` などのアサーションは、条件が成立するまで再試行します。そのため、このテスト仕様に固定の待ち時間はありません。`page.route` は、ブラウザーから配送業者へのリクエストに、固定の本文で応答します。チームが運用しないサーバーに、テストが依存しなくなります。globパターンはURL全体と一致する必要があり、`**` はスラッシュをまたいで一致します。ルーティングが扱うのは、ページ自体が送るリクエストだけです。店のサーバーが配送業者へ問い合わせる場合、そのリクエストはブ

ラウザーに見えないため、サーバー側へテストダブルを置く必要があります。各テストには、新しいブラウザーコンテキストも与えます。Playwrightは、これを作りたてのブラウザープロファイルに例えます。一つのテストのCookieや買い物かごが、次へ漏れることはありません。

設定は、アプリを起動し、その場所をPlaywrightへ伝えます。

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

`webServer` ブロックは `npm run start` を実行し、最初のテストの前に、URLが応答するまで待ちます。ノートパソコンでは、すでに動くサーバーを再利用し、CIでは、必ず専用のサーバーを起動します。 `baseURL` により、完全なアドレスではなく、 `/shop/teapot` と書けます。 `npx playwright test` はウィンドウを表示せずにテスト群を実行し、 `--headed` はブラウザーのウィンドウを表示します。 `npx playwright show-report` は、HTMLレポートを開きます。Playwrightは、既定では、失敗したテストを再試行しません。この設定では、CIで二回の再試行を許します。一度失敗してから通ったテストは、合格ではなく、不安定と報告します。最初の再試行でトレースを記録し、何が違ったかを示します。

規模が大きい場合、作業全体に占めるブラウザー層の割合は、最も小さくなります。2025年からの職務で手がけたGo、SQL、ブラウザーをまたぐテスト群には、310ファイルに981個のGoテスト関数、VitestとPlaywrightの543個のテスト仕様、その中に48個のエンドツーエンド用ファイルと30個のブラウザー用テスト仕様、さらにSQLの振る舞いを試す494個のファイルがあります。課題は、すべてを外側のブラウザーから試すのではなく、各層の振る舞いを、その層で確か

めることでした。ストアド関数なら、その上のアプリの三層を通さず、データベースで試します。事例は、負担も明記しています。テスト群は遅く、すべての変更に負担を加え、その規模では、テスト群自体の保守も必要です。

問題が起きるところ

テスト群が、アイスクリームのコーン型です。 エンドツーエンドテストが多く、結合テストと単体テストが少ない構成を、Benderはこう呼びます。通常、遅く、不安定で、扱いにくい構成です。試作から本番へ急いで進んだプロジェクトによく見られます。Gravesは、単体テストが難しいコードでは、書けるのがエンドツーエンドテストだけになり、数日のうちにレガシーコードを作ってしまうと補足します。サービスが増えるたびに、システム内の経路も掛け算で増えるため、エンドツーエンドのテスト群ですべては網羅できません。各ケースは、それを検出できる最も小さいテストで確かめます。ブラウザーを使うのは、システム全体でなければ確かめられない一連の操作に限り、プロジェクトの初期からピラミッドへ近づけます。

セレクターが、マークアップに依存します。 Playwright自身が避ける例に挙げる `button.buttonIcon.episode-actions-later` のようなロケーターは、クラス名を変えるだけで壊れます。利用者には何も違って見えなくても同じです。Gravesは、UIテストには不安定でコストが高いという評判があり、振る舞いが同じでも見た目の変更で壊れることが、その一因だと警告します。利用者と同じように、ロール、ラベル、文字列か、チームが管理するテストIDで要素を探します。バックエンドに公開APIがあるなら、UIとAPIの境界で操作の流れを分け、残りのエンドツーエンドテストは、APIを通して実行します。

テストが、決まった時間だけ眠ります。 固定の待ち時間は、遅い日には短すぎ、ほかの日には長すぎます。Gravesは、固定の待機やタイムアウトに頼るテストは、実行マシンに過負荷がかかると一斉に失敗し始め、再実行で負荷がさらに増えると述べます。Playwrightのベストプラクティスは、より見つけにくい同じ誤りを示します。 `expect(await page.getByText("welcome").isVisible()).toBe(true)` は、一回調べて先へ進みます。一方、 `await expect(page.getByText("welcome")).toBeVisible()` は、条件を待ちます。ウェブの状態を待つアサーションを使い、バックエンドを待つテストでは、タイムアウトを設けて状態の変化をポーリングします。

テスト間で、データを共有します。 一つのテストが作るアカウントを、次が当てにしたり、並列の二つのワーカーが、在庫の最後のティーポットを買ったりします。ある順序では通り、別の順序や並列実行時だけ失敗します。Playwrightは、各テストが、それぞれの保存領域、セッション、データ、Cookieを持ち、完

全に隔離されるよう勧めます。テストが使うデータベースのデータも、制御します。各テストが必要とするものは、一意の名前で作り、前のテストが残したものに頼りません。

他者のサーバーを、テストに含めます。 本物の決済事業者、配送業者、地図サービスを含めると、チームが管理できないシステムへ依存します。トラフィックの送信に、費用がかかることもあります。Gravesは、自動テストから本物の外部APIを呼ばないよう勧め、そうした依存先を、テストを分割する重要な境界と呼びます。Playwrightも、管理できるものだけを試すよう勧めます。上の例のように、ブラウザーが呼ぶなら `page.route` で、サーバーが呼ぶなら、そのサーバーの呼び出し先に設定したテストダブルで、外部の相手に代わって応答します。その依存先を、テストの終点にします。

再試行が、不安定さを隠します。 CIの再試行は、赤いビルドを緑にし、負担が積み重なります。Benderは、計算で示します。一日に10,000件のテストで、それぞれの誤検出による失敗率が0.1パーセントなら、毎日10件の不安定な失敗を調べる必要があります。不安定な失敗率が1パーセントへ近づくと、テストは価値を失い始めます。再実行は、エンジニアの時間を節約する代わりにマシンの時間を使い、根本原因を先送りするだけです。pytestの文書は、別の角度から同じ点を説明します。上位のテストほど、多くの状態に依存するため、断続的に失敗しやすくなります。Playwrightが不安定と報告したテストは、すべてバグとして扱い、再試行のトレースを開き、直すか削除します。

参考資料

- テストの概要 — _Software Engineering at Google_のAdam Benderによる章です。テストのサイズと範囲、ピラミッド、アイスクリームのコーン型、不安定なテストの負担を扱います。
- 大きいテスト — Joseph Gravesによる、忠実度、単体テストに残る穴、UIや外部サービスとの境界、不安定さの除去です。
- インストール — Playwright Testのインストール、テスト例の実行、ウィンドウを表示するモード、HTMLレポートについての、Playwright自身の導入文書です。
- ベストプラクティス — 利用者に見える振る舞いのテスト、テストの隔離、外部サーバーの除外、ロケーター、ウェブの状態を待つアサーションです。
- 再試行 — Playwrightが失敗したテストを再試行し、合格、不安定、失敗に分類する方法です。
- 不安定なテスト — 制御されていないシステムの状態、テスト順序、スレッドセーフティーなど、pytestが説明する不安定な失敗の原因と、その対処です。

用語集

実際の制約に合わせて選ぶ

業務の流れ、既存のインターフェース、変更を必要とする利用者をお知らせください。検証可能な納品内容と、お客様のチームが保守を続けられる引き継ぎ内容を定められます。

料金を見る お問い合わせ →

用語を実務に結び付ける

「エンドツーエンドテストとは？」がお客様の組織にどう当てはまるかをご相談ください。現在のシステムの具体例が一つあれば、話を始められます。

サービスを見る

welcome@engineer.company +45 50 10 61 84 デンマーク、コペンハーゲン

このページを持ち歩く

連絡先カード

心を込めて丁寧に制作： D.

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

著作権 © 2025 – 現在 · Engineer ApS · すべての権利を留保します。

印刷元： https://engineer.company/ja/glossary/end-to-end-test/