Playwright のあるきかた

～E2E テストの導入から CI 構築まで～

ゆずねり　著

2024-11-02 版　　NeIn 発行

はじめに

近年、ウェブアプリケーションはますます重要性を増しており、開発の効率化と品質向上のためには E2E テストが欠かせなくなってきました。本書は 2020 年と比較的最近リリースされた、Playwright の使い方を解説します。

前著「Playwright のあるきかた ～ゼロから始める E2E テスト～」では入門編として、テストをするうえで重要なロケーターやマッチャーの基本を解説しました。

本書では実践編として、ユニットテストと Playwright の付き合い方から始まり、テストコードの共通化やクッキーの共有、通信の監視など、より Playwright を活用する方法の解説をします。

本書を通じて、Playwright の知識が深まれば幸いです。

対象読者

本書の対象読者は、Playwright を触ったことがあり、より詳しいことを知りたいと思っている方向けになります。

もし Playwright をまだ触ったことがない方は、まずは前著「Playwright のあるきかた ～ゼロから始める E2E テスト～」からご覧ください。

前提知識

本書はウェブサイトの E2E テストを目的としているため、最低限の HTML の知識が必要になります。

本書内のコードは、TypeScript で書かれています。難しい処理は行っていないため、他言語の経験がある方であれば問題なくお読みいただけると思います。

動作環境

本書の内容は、次の環境で確認をしています。

- Windows Subsystem for Linux 上の Ubuntu v22.04 LTS
- macOS v15
- Docker v27.2.0
- Node.js v22.9.0
- Playwright v1.48

本書で使用するサンプルコード

本書では、次のサンプルコードを用いて解説します。
`https://github.com/yuzneri/PlaywrightGuide2SampleCode`

本書サポートサイトのご案内

本書のサポートサイトは次の URL になります。
`https://neln.net/b/nt02/`

免責事項

本書に記載された内容は、情報の提供のみを目的としています。したがって、本書を用いた開発、製作、運用は、必ずご自身の責任と判断によって行ってください。これらの情報による開発、製作、運用の結果について、著者はいかなる責任も負いません。

目次

第 1 章

テストの考えかた

本章では、各テスト手法の役割と、Playwright を使ったテストの効率化について解説します。

各テスト手法の役割を理解したうえで活用すると、開発プロセスの効率化や信頼性の向上が期待できます。

1.1　テスト手法

ソフトウェア開発では主にユニットテスト、インテグレーションテスト、E2E テストの 3 つの手法があります。

1.1.1　ユニットテスト（単体テスト）

ユニットテストは、関数やメソッドなどの最小単位で正しく動作するか確認するテストです。モックやスタブを用いてデータベースなどの副作用がない、独立した状態でテストします。

比較的容易に自動化でき、高速にテストできるため、開発プロセスの中で頻繁に行われます。これにより、コードの品質を向上させ、不具合の早期発見が可能です。

1.1.2　インテグレーションテスト（結合テスト）

インテグレーションテストは、複数のユニットが組み合わさったときに正しく相互作用するか確認するテストです。ユニットテストとは異なり、データベースなどの副作用も含めた状態でテストします。

ユニットテストでみつけられないインターフェース間の問題や、データの不整合を発見

できます。

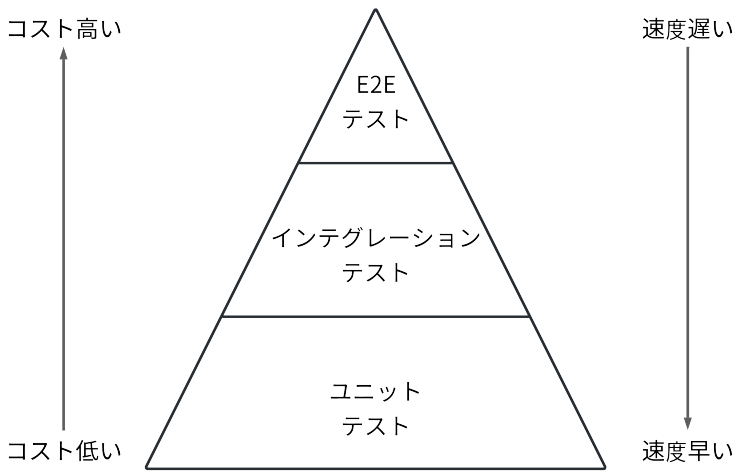
1.1.3　E2E テスト

E2E（End-to-End）テストは、ユーザーの視点からシステム全体が正しく動作するか確認するテストです。実際のユーザー環境に近い条件で、想定されるユーザーシナリオに基づいてテストします。

フロントエンド、バックエンド、データベースなどソフトウェア全体が期待どおりに動作するか確認します。これによりシステム全体の統合性を確認でき、ユーザーエクスペリエンスを向上できます。

1.2　テストピラミッド

テストピラミッドは、Mike Cohn 氏が提唱した概念です。ユニットテスト、インテグレーションテスト、E2E テストの関係を図 1.1 のようなピラミッドで表したものです。



▲図 1.1　テストピラミッド

下位のテストでは、テストケースが多く安定して速くなるものの、実際の挙動からは離れています。上位のテストでは、テストケースが少なく不安定[*1]で遅くなるものの、実際の挙動に近づきます。

[*1] 変更がないにもかかわらず成功したり失敗したりするテストを、不安定なテストと言います

第 2 章

テストテクニック

本章では、Playwright でテストをするうえで遭遇する、認証やネットワーク周りなどのテストテクニックを解説します。

2.1 テストジェネレーターでテストコードを作ってみよう

すでにウェブサイトが稼働しているのであれば、Playwright 内蔵のテストジェネレーターでテストコードを作成するのが簡単です。今回は、サンプルアプリケーションのトップページにアクセス、ログインのテストを作成してみます。

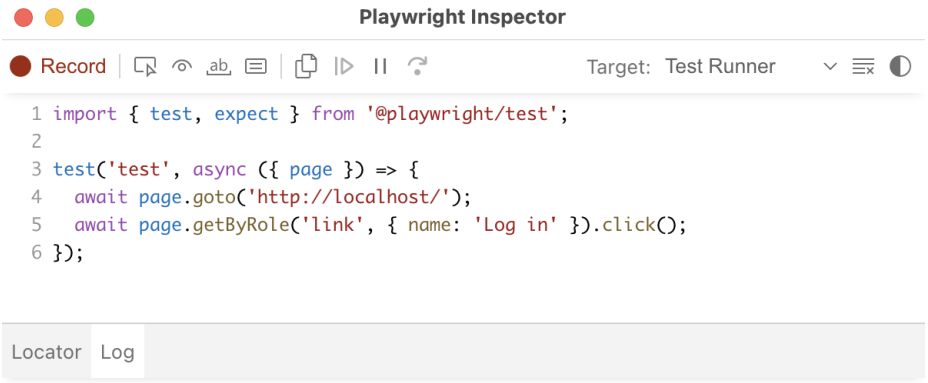
2.1.1 ウェブブラウザを操作するコードの作成

テストジェネレーターは、次の `playwright` コマンドで起動できます。

```
npx playwright codegen
```

コマンドを実行すると、ウェブブラウザと Playwright Inspector が起動します。

記録が開始されているので、ウェブブラウザのアドレスバーに `http://localhost/` を入力しましょう。ウェブページが表示されるとともに、インスペクターにページ遷移のコードが追加されます。そのまま右上の Log in リンクをクリックすると、次の行にリンククリックのコードが追加されます。



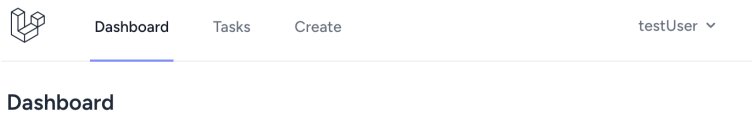
▲図 2.1　Playwright Inspector に記録される様子

このように、ウェブブラウザを普段どおりに操作することで、自動的にテストコードが作成されます。注意点として、関係のないマウスクリックやキーボード入力なども記録されることがあります。あとで大変なことになるので、余計な操作は極力控えましょう。

ログインページでは、メールアドレス `test@example.com`、パスワード `test` を入力し、Log in ボタンを押します。

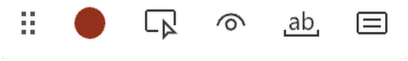
2.1.2　画面を検証するコードの作成

ログインに成功するとダッシュボードページが表示されます。



▲図 2.2　ダッシュボードページ

ヘッダーにユーザー名が表示されているので、正しいアカウントでログインが行われているか検証してみましょう。ウェブブラウザ、あるいはインスペクター上のツールバーから検証コードを追加できます。

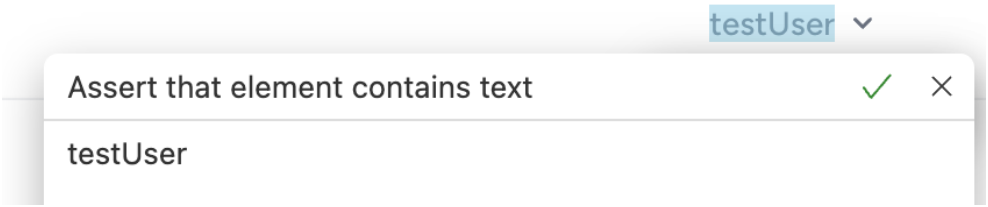


▲図 2.3　ツールバー

ツールバーでは、次のような操作ができます。

- Record
  - 操作の記録、停止
- Pick locator
  - 選択した要素のロケーターを表示
- Assert visibility
  - 選択した要素が表示されているか検証するコードを追加
- Assert text
  - 選択したテキスト要素の文字列が正しいか検証するコードを追加
- Assert value
  - 選択したフォーム要素の文字列が正しいか検証するコードを追加

今回はテキスト要素を検証したいので、「Assert text」ボタンをクリックします。検証したい要素をマウスオーバーすると、その要素の背景が青くなります。そのままクリックすると、検証したい文字列を入力するフォームが表示されます。デフォルトで選択した要素のテキストが入力されているので、そのままチェックマークをクリックすると検証コードが追加されます。



▲図 2.4　テキスト要素の検証

2.1.3　できたテストコードでテスト

ここまでで、インスペクター上には次のようなテストコードができています。

▼リスト 2.1　できあがったテストコード

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

test('test', async ({page}) => {
  await page.goto('http://localhost/');
  await page.getByRole('link', {name: 'Log in'}).click();
  await page.getByLabel('Email').click();
  await page.getByLabel('Email').fill('test@example.com');
  await page.getByLabel('Email').press('Tab');
  await page.getByLabel('Password').fill('test');
  await page.getByRole('button', {name: 'Log in'}).click();
  await expect(page.getByTestId('loginId')).toContainText('testUser');
});
```

このままでもテストできますが、フォームをクリックするコード（.click()）や、TABキーを押すコード（.press('Tab')）などのコードも含まれています。フォーム入力だけであれば、これらのコードは不要です。整理すると、最終的に次のようになります。

▼リスト 2.2　ログインテスト

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

test('test', async ({page}) => {
  await page.goto('http://localhost/');
  await page.getByRole('link', {name: 'Log in'}).click();
  await page.getByLabel('Email').fill('test@example.com');
  await page.getByLabel('Password').fill('test');
  await page.getByRole('button', {name: 'Log in'}).click();
  await expect(page.getByTestId('loginId')).toContainText('testUser');
});
```

テストディレクトリに保存し、テストコマンドを実行するとテストできます。

```
npx playwright test
```

このようにウェブサイトがあるときは、テストジェネレーターを使うと簡単にテストコードを作成できます。ただし、テストジェネレーターだけですべての操作や検証を網羅することはできません。必要に応じて、自分でテストコードを書く必要があります。

第 3 章

CI でも Playwright

本章では、Playwright を CI（継続的インテグレーション）で実行する方法を解説します。

CI に組み込むことにより、push や Pull request のタイミングで自動的にテストが行われ、不具合の早期発見につながります。

3.1 Playwright が対応する CI サービス

Playwright は次の CI サービスに対応しています。

- GitHub Actions
- Azure Pipelines
- CircleCI
- Jenkins
- Bitbucket Pipelines
- GitLab CI
- Google Cloud Build
- Drone

また、ここにないサービスでも CLI でコマンドを自由に実行できる環境、もしくは Docker が動く環境であれば問題ありません。

3.2 GitHub Actions で Playwright

ここからは、GitHub Actions を例に説明します。

第 4 章

Playwright の設定

本章では、Playwright の設定について説明します。

Playwright にはテストの実行方法に関するオプションが多数あり、これらを細かく設定できます。

4.1 Playwright の設定方法

Playwright の設定は、`playwright.config.ts` ファイルで変更できます。テストに利用するウェブブラウザや、テスト時の挙動などを設定できます。

4.1.1 環境によって設定を使い分ける

Playwright 自体には、環境ごとに設定を分ける仕組みは用意されていません。しかし、設定ファイルは TypeScript で書かれているため、カスタマイズが可能です。dotenv を使うコードがコメントアウトされているので、こちらを使うのがよいでしょう。

まず、`playwright.config.ts` ファイルと同じディレクトリに`.env` ファイルを作成し、キー=バリューのセットで設定を用意します。

▼リスト 4.1　.env に設定を書く

```
USE_VIDEOS=on
TIMEOUT=45000
```

次に、dotenv モジュールをインストールします。

```
npm install dotenv --save
```

playwright.config.ts ファイルから、dotenv.config() 部分をアンコメントします。

▼リスト 4.2　dotenv を読むコード

```
import dotenv from 'dotenv';
import path from 'path';
dotenv.config({path: path.resolve(__dirname, '.env')});
```

process.env 経由でキーを指定できます。

▼リスト 4.3　dotenv の設定を使う

```
export default defineConfig({
  timeout: Number(process.env.TIMEOUT) ?? 30000,
  use: {
    video: process.env.USE_VIDEOS ?? 'off',
  },
});
```

4.2　基本オプション

タイムアウトや、リトライ、ワーカープロセス数などテスト全般の設定ができます。

4.2.1　テスト全体のタイムアウト

テスト全体のタイムアウトを、globalTimeout プロパティで設定できます。デフォルトは無限です。

▼リスト 4.4　テスト全体のタイムアウトを設定

```
export default defineConfig({
  globalTimeout: 600000,
});
```

4.2.2 各テストのタイムアウト

フィクスチャやフックを含む各テストのタイムアウトを、timeout プロパティで設定できます。デフォルトは 30,000 ミリ秒です。

▼リスト 4.5　テストのタイムアウトを設定

```
export default defineConfig({
  timeout: 60000,
});
```

一部のテストだけ時間がかかる場合は、個別に設定することもできます。

▼リスト 4.6　個別にタイムアウトを設定

```
test('slow test', async ({page}) => {
  // 設定の3倍
  test.slow();
});

test('very slow test', async ({page}) => {
  test.setTimeout(120000);
});
```

4.2.3 アサーションのタイムアウト

アサーションのタイムアウトを、expect.timeout プロパティで設定できます。デフォルトは 5,000 ミリ秒です。

▼リスト 4.7　テスト全体にアサーションタイムアウトを設定

```
export default defineConfig({
  expect: {
    timeout: 10000,
  },
});
```

一部のアサーションだけ時間がかかる場合は、個別に設定することもできます。

▼リスト 4.8　個別にアサーションタイムアウトを設定

```
await expect(locator).toBeVisible({timeout: 10000});
```

4.2.4　リトライ回数

テスト失敗時の最大試行回数を、retries プロパティで設定できます。デフォルトは再試行しません。

▼リスト 4.9　リトライ回数を設定

```
export default defineConfig({
  retries: 3,
});
```

4.2.5　テストの並列実行

テストを並列に実行するかどうかを、fullyParallel プロパティで設定できます。また、並列で実行する場合は workers プロパティで、ワーカーの最大プロセス数を設定できます。

▼リスト 4.10　並列数を設定

```
export default defineConfig({
  fullyParallel: true,
  workers: 3,
});
```

4.2.6　テスト状況の記録

テスト時のスクリーンショット、ビデオ、トレースを記録できます。デフォルトは記録されません。

▼リスト 4.11　テスト状況の記録条件を設定

```
export default defineConfig({
  use: {
    screenshot: 'only-on-failure',
    video: 'on-first-retry',
```

```
    trace: 'on-first-retry',
  },
});
```

スクリーンショット

screenshot プロパティで、スクリーンショットの撮影を制御します。

- off：保存しない
- on：すべて保存する
- only-on-failure：失敗したテストだけ保存する

ビデオ

video プロパティで、ビデオの録画を制御します。

- off：保存しない
- on：すべて保存する
- retain-on-failure：失敗したテストだけ保存する
- on-first-retry：リトライしたテストだけ保存する

トレース

trace プロパティで、トレースの記録を制御します。

- off：保存しない
- on：すべて保存する
- retain-on-failure：失敗したテストだけ保存する
- on-first-retry：リトライしたテストだけ保存する

トレースは各アクションごとに利用したロケーターやかかった時間を、DOM のスナップショットやスクリーンショットなどと一緒に確認できます。

トレースは、次のコマンドで確認できます。ローカルかリモートの zip ファイルを指定します。

```
npx playwright show-trace path/to/trace.zip
npx playwright show-trace https://example.com/trace.zip
```

Playwright のあるきかた
〜E2E テストの導入から CI 構築まで〜

2024 年 11 月 2 日　初版第 1 刷　発行

発行所　Neln
印刷所　日光企画