Playwright のあるきかた

～ゼロから始める E2E テスト～

ゆずねり　著

2024-05-25 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。

Playwright は JavaScript で記述されたコードを実行することで、Chrome、Firefox、Safari を自在に操作できます。ウェブブラウザを自動操作するツールとして、シンプルさと高い信頼性、コードの可読性を兼ね備えています。

本書では Playwright をはじめてさわる方が、導入から E2E テストを書けるようになるまでの基本的な使い方を解説します。

対象読者

本書の対象読者は、これから Playwright を触ってみようと思っている方向けになります。

Playwright の導入、基本的な API の解説を行い、ひととおりの E2E テストが書けるようになることを目標としています。

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、JavaScript 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 v14
- Docker v26.0.0
- Node.js v20.12.1
- Playwright v1.44.0

本書で使用するサンプルアプリとサンプルコード

本書では、次のサンプルアプリを用いて解説を行います。
`https://yuzneri.github.io/playwrighttodolist/`
本書で使用したコードの全文は、GitHub に公開しています。
`https://github.com/yuzneri/PlaywrightGuideSampleCode`

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/nt01/`

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

ウェブブラウザを自動操作する技術

本章では、E2E テストを実現するためのツールについて解説します。

ウェブブラウザを使って E2E テストを行う必要性と、長い歴史の中で誕生したさまざまなツールの紹介を行います。

1.1　なぜウェブブラウザを自動操作するのか

ウェブサイトはユーザーがウェブブラウザの操作することを前提に、開発していることが多いと思います。

ウェブブラウザを自動操作できれば、実際のエンドユーザーと同じ視点で動作検証ができます。近年、ウェブブラウザの自動操作を支援するツールが大きく発展したことにより、ウェブアプリケーションのテスト自動化が容易になってきました。

その結果、フロントエンドとバックエンドの統合テストを効率的に実施できるようになりました。また、CI/CD パイプラインに組み込むことで自動的に、さまざまなウェブブラウザ、バージョン、デバイスでの回帰テストが容易になり品質の向上に大きく貢献します。

つまり、ウェブブラウザの自動操作は、エンドユーザーに近い視点から品質を確保するための重要な手段になります。

1.2　ウェブブラウザの種類

ウェブブラウザには大きく分けて、フルブラウザとヘッドレスブラウザの 2 種類があります。

フルブラウザは皆さんが普段使っている、GUI を持った一般的なウェブブラウザです。

ユーザーがマウスやキーボードで直接ウェブブラウザを操作してウェブページを閲覧できます。

ヘッドレスブラウザは GUI を持っていなく、ユーザーはマウスやキーボードで直接操作することはできません。GUI を持っていないだけで通常のフルブラウザ同様、HTML や JavaScipt を解釈、レンダリングします。マウスやキーボードの代わりにプログラミングで、レンダリングされたウェブサイトを操作できます。

1.3 ウェブブラウザを操作するツール

ウェブブラウザを自動操作するためのさまざまなツールがあります。ここでは代表的な 4 つのツールについて簡単に説明します。

Selenium

Selenium[*1]は 2004 年にリリースされたツールです。ウェブ標準規格の WebDriver を使い、Java や Python、JavaScript など多くの言語で主要ウェブブラウザを操作できます。

多くのウェブブラウザやプラットフォームに対応、ドキュメントも充実している一方、WebDriver が対応していないウェブブラウザ固有機能には制約があります。

PhantomJS

PhantomJS は、2011 年にリリース、2018 年に事実上の開発が終了したツールです。

ヘッドレスブラウザの先駆けとして、ウェブブラウザ自動化の発展に大きく貢献しました。

Cypress

Cypress[*2]は Cypress 社が主導し、2017 年にリリースされた JavaScript ベースのツールです。オープンソースですが一部の機能は有料です。他のツールと違い、ウェブページに JavaScript を埋め込んで Chrome、Firefox を操作します。実際のユーザー環境に近い状態で操作できる一方、複数タブや複数ドメインにまたがるケースには向いていません。

[*1] https://www.selenium.dev
[*2] https://www.cypress.io

Puppeteer

Puppeteer[*3]は Google が主導し、2018 年にリリースされた Node.js 用のツールです。Chrome DevTools Protocol で Chrome を操作することにより、Selenium よりも細かく制御できます。

元々は Chrome 用でしたが、あとに WebDriver BiDi に対応したことにより Firefox でも使えます。

Playwright

Playwright[*4]は Microsoft が主導し、2020 年にリリースされたツールです。初期の Puppeteer を開発していたチームによって開発されているため、概念や API は Puppeteer と似ています。

Puppeteer と比べると Node.js のほかに、Python、.NET、Java に対応しています。操作できるウェブブラウザは Chromium のほか、Firefox[*5]、WebKit[*6]に対応しています。

ウェブサイトから見たフルブラウザとヘッドレスブラウザの違い

使う側からするとフルブラウザとヘッドレスブラウザは、GUI の有無や入力の違いがあると解説しました。では、ウェブサイト側での違いはあるのでしょうか。

結論から書くと、JavaScript の挙動にいくつかの違いがあります。Chrome 使用時の違いを 2 つご紹介します。

navigator.userAgent

`navigator.userAgent` は、ウェブブラウザのユーザーエージェントを取得します。ヘッドレスブラウザで取得すると、HeadlessChrome と表記されます。

[*3] `https://pptr.dev`
[*4] `https://playwright.dev`
[*5] 厳密には最新の Firefox Stable ビルドにパッチ適用している
[*6] 厳密には最新の WebKit trunk ビルドにパッチ適用している

▼表 1.1　User agent の違い

動作方法	出力
フルブラウザ 人間	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/124.0.0.0 Safari/537.36
フルブラウザ Playwright	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/124.0.0.0 Safari/537.36
ヘッドレスブラウザ Playwright	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/124.0.6367.29 Safari/537.36

navigator.webdriver

`navigator.webdriver` は、ヘッドレスブラウザかどうかは関係なく、ウェブブラウザが自動化されていると true になります。

▼表 1.2　User agent の違い

動作方法	出力
フルブラウザ 人間	false
フルブラウザ Playwright	true
ヘッドレスブラウザ Playwright	true

そのほかの違い

GUI の有無、入力方法の違いなどから判定できる場合もあります。

詳しくは Fingerprint Scanner[*7]などを参考にしてください。

[*7] `https://github.com/antoinevastel/fpscanner`

第 2 章

Playwright のインストール

本章では、Playwright をセットアップする手順について解説します。

Node.js 環境に対して Playwright をインストールする方法と、Docker コンテナ上で Playwright を実行する方法の解説をします。

2.1 ネイティブ環境で使用する場合

ネイティブ環境では前提条件として、Node.js バージョン 18 以上のインストールが必要です。Node.js がインストールされていれば、`npm init` コマンドで Playwright のインストールができます。

```
npm init playwright@latest
```

インストール中いくつかのオプションを聞かれます。

Do you want to use TypeScript or JavaScript?

TypeScript と JavaScript のどちらを使うか。デフォルトは TypeScript です。

Where to put your end-to-end tests?

E2E テストのプログラムを保存するディレクトリ名。デフォルトは tests、すでに tests がある場合は e2e になります。

Add a GitHub Actions workflow?

GitHub Actions ワークフローを追加するか。デフォルトは追加しません。

Install Playwright browsers?

Playwright が使うウェブブラウザをインストールするか。デフォルトはインストールします。

あとからインストールするには、`install` コマンドを使用します。

```
npx playwright install
```

▼表 2.1　ウェブブラウザのインストール先

プラットフォーム	インストール先
Windows	%USERPROFILE%\AppData\Local\ms-playwright
MacOS	~/Library/Caches/ms-playwright
Linux	~/.cache/ms-playwright

Install Playwright operating system dependencies?

システム依存関係をインストールするか。デフォルトはインストールしません。

パッケージマネージャーを使用して画像や動画などのマルチメディア関連のライブラリ、フォントライブラリなどがインストールされます。そのため root 権限が必要です。

あとからインストールするには、`install-deps` コマンドを使用します。

```
sudo npx playwright install-deps
```

インストール時に作成されるファイル

- playwright.config.js
  - テスト構成ファイル
- tests/example.spec.js
  - テストサンプルコード
- tests-examples/demo-todo-app.spec.js
  - ToDo アプリを想定した、より具体的なテストサンプルコード

2.2 Docker 環境で使用する場合

公式 Docker イメージは、Microsoft Artifact Registry に公開されています[*1]。このイメージにはウェブブラウザと、システム依存関係が含まれています。

`docker pull` コマンドでイメージを pull します。

```
docker pull mcr.microsoft.com/playwright:latest
```

Playwright 本体は含まれていないため、「2.1 ネイティブ環境で使用する場合」を参考に別途インストールしてください。ウェブブラウザとシステム依存関係はイメージに含まれているので、インストールする必要はありません。

準備が整ったら、`docker run` コマンドで実行できます。

```
docker run -it --rm --ipc=host \
  -v ${PWD}:/app mcr.microsoft.com/playwright:latest node app/test.js
```

2.3 Playwright のアップデート

Playwright 本体のアップデートは、次のコマンドになります。

```
npm install -D @playwright/test@latest
```

ウェブブラウザとシステム依存関係のアップデートは、次のコマンドになります。

```
sudo npx playwright install --with-deps
```

[*1] https://mcr.microsoft.com/en-us/product/playwright/about

2.4 実行テスト

インストール時に作成されるサンプルテストを動かし、正しくインストールできたか確認してみましょう。

`playwright test` コマンドでテストが実行できます。

```
npx playwright test
```

正しくインストールできていれば、次のような結果になります。

```
Running 6 tests using 6 workers
  6 passed (9.0s)
```

第 3 章

Playwright API

本章では、Playwright API の主要な機能について解説します。

Playwright を使ったウェブブラウザの操作方法、スクリーンショットの撮影など、Playwright API で提供される自動操作の方法を取り上げます。

3.1 Playwright の始め方

Playwright を扱うときの全体の流れは、次のようになります。

1. ウェブブラウザを起動
2. 目的のページを開く
3. ページ内の要素に対して処理
4. ウェブブラウザを終了

この処理をコードに書くと、次のとおりになります。

▼リスト 3.1　初期化処理と終了処理

```
const {chromium, firefox, webkit} = require('playwright');

(async () => {
  // 1. ウェブブラウザを起動する
  const browser = await chromium.launch();

  // 2. 目的のページを開く
  const page = await browser.newPage();
  const response = await page
    .goto('https://yuzneri.github.io/playwrighttodolist/');

  // 3. ページ内の要素に対して処理
```

```
  await page.getByPlaceholder('やること').fill('カレーを作る');
  await page.getByRole('button', {'name': '追加'}).click();

  // 4. ウェブブラウザを終了する
  await browser.close();
})();
```

3.1.1 ウェブブラウザを起動

最初にブラウザインスタンスを launch() で作成します。Chromium は chromium.launch()、FireFox は firefox.launch()、Webkit は webkit.launch() を使うことによりヘッドレスブラウザを起動できます。

▼リスト 3.2　各種ウェブブラウザの起動

```
// Chrome
const browser = await chromium.launch();

// FireFox
const browser = await firefox.launch();

// Webkit
const browser = await webkit.launch();
```

フルブラウザを起動する場合は、headless プロパティを false に指定します。主に動作チェック目的で使用します。

▼リスト 3.3　フルブラウザを起動

```
const browser = await chromium.launch({headless: false});
```

3.1.2 ページを作成

次に新しいページ（いわゆるタブ）を、newPage() で作成します。このページを元に、いろいろな操作をしていきます。

▼リスト 3.4　ページを作成

```
const page = await browser.newPage();
```

3.1.3　ページを移動

対象のウェブサイトに goto() で移動します。リダイレクトがある場合は、最後までリダイレクトされます。URL は https://などのスキームから指定する必要があります。

▼リスト 3.5　ページを移動

```
await page.goto('https://yuzneri.github.io/playwrighttodolist/');
```

次のようなときは例外が発生します。

- SSL エラー
- URL が無効
- タイムアウト
- サーバーに到達できない

一方、400 番台のクライアントエラーや、500 番台のサーバーエラーなどは例外にはなりません。

3.1.4　ページ内の要素に対して処理

ヘッドレスブラウザでは、マウスやキーボードでの入力ができません。なんらかの方法でマウスやキーボードの代わりにページ内の要素をみつけ、テキスト入力やクリックなどのアクションを行う必要があります。

ElementHandle

ElementHandle は$() や、$$()、waitForSelector() などのメソッドを使用し、ページの要素に対して処理を行います。もともとは Puppeteer 用の機能であり、Playwright での使用は推奨されていません。代わりに Locator を使用してください。

Locator

ElementHandle 同様、ページの要素に対して処理を行います。テキストや、フォームに設定されたプレースホルダーのテキストやなど、状況に応じたさまざまなメソッドが用意されています。また、ElementHandle にはない、自動待機や再試行を備えています。

3.1.5　ウェブブラウザを閉じる

最後にウェブブラウザを、close() で閉じる必要があります。ウェブブラウザを閉じると、Browser オブジェクトは破棄され使用できなくなります。

```
await browser.close();
```

3.2　レスポンス

3.2.1　すべてのヘッダー情報を取得する

ヘッダー情報をオブジェクトで取得するには allHeaders()、配列で取得するには headersArray() を使います。

▼リスト 3.6　すべてのヘッダー情報を取得

```
await response.allHeaders();
await response.headersArray();
```

3.2.2　特定のヘッダー情報を取得する

特定のヘッダー情報を配列で取得するには headerValues()、コンマ区切りの文字列で取得するには headerValue() を使います。

指定するヘッダー名の大文字小文字は区別されません。

▼リスト 3.7　特定のヘッダー情報を取得

```
await response.headerValue('set-cookie');
await response.headerValues('set-cookie');
```

3.2.3　HTTP ステータスを取得する

応答が成功したかどうか（200〜299 の範囲内）を確認するには、ok() を使います。

詳しいステータスコードを確認するには status()、ステータステキストを確認するには statusText() を使います。

▼リスト 3.8　HTTP ステータスを取得

```
response.ok();
response.status();
response.statusText();
```

3.2.4　HTML 全文を取得する

HTML 全文を取得するには、text() を使います。JSON 形式であれば json() を使うことにより、パースした結果を取得できます。

▼リスト 3.9　本文を取得

```
await response.text();
await response.json();
```

3.3　Locator

Locator は Playwright がページ上の要素をみつけるための方法です。アクセシビリティ属性や、フォームラベルなどさまざまな手段があります。

Locator は要素をみつけられるまで、自動的に待機します。そのため、目的の要素がレンダリングされるまで待つ処理を書く必要はありません。

3.3.1　ロールで探す

HTML には ARIA ロールや属性[*1]、アクセシブル名[*2]がそれぞれ定義されています。アクセシブル属性を使ってロールを探すには、getByRole() を使います。第一引数に ARIA ロール名、第二引数にオプションを指定します。

ARIA ロールには、ボタン、チェックボックス、見出し、リンク、テーブルなどが含まれています[*3]。ユーザーがページを認識するのに一番近い方法であり、Playwright では

[*1] https://developer.mozilla.org/ja/docs/Web/Accessibility/ARIA
[*2] https://developer.mozilla.org/ja/docs/Glossary/Accessible_name
[*3] https://developer.mozilla.org/ja/docs/Web/Accessibility/ARIA/Roles

一般的な方法のため、通常はこの Locator の使用をおすすめします。

次の DOM 構造について考えてみましょう。

```
<label for="new-todo">新規タスク</label>
<input id="new-todo">
<button id="todo-add">追加</button>
```

要素を探し、タスクの追加ボタンを押すには次のようなコードを書きます。

▼リスト 3.10　フォームにテキストを設定し、ボタンを押す

```
await page.getByRole('textbox', {'name': '新規タスク'}).fill('カレーを作る');
await page.getByRole('button', {'name': '追加'}).click();
```

設定する ARIA ロールやアクセシブル名は、Chrome デベロッパーツールから確認できます。調べたいタグを選択し、アクセシビリティタブの計算済みプロパティから確認できます。

第 4 章

Playwright E2E テスト

本章では、Playwright を使った E2E テストの実装方法について解説します。

テストコードの基本的な記述方法に加え、E2E テスト作成に役立つさまざまな機能を取り上げます。

4.1　テストの始め方

Playwright で E2E テストを始める、最低限のコードは次のようになります。

▼リスト 4.1　最低限の E2E テストコード

```
const {test, expect} = require('@playwright/test');

test('タスクの追加', async ({page}) => {
  // テストしたいページに移動
  await page.goto('https://yuzneri.github.io/playwrighttodolist/');

  // テストしたい操作を行う
  await page.getByPlaceholder('やること').fill('カレーを作る');
  await page.getByRole('button', {'name': '追加'}).click();

  // テストしたい要素を選択
  const locator = page.getByRole('listitem').last().locator('.todo-text');

  // テストを行う
  await expect(locator).toHaveText('カレーを作る')
});
```

「3.1 Playwright の始め方」では `chromium.launch()` と `browser.newPage()` を使って、`page` を用意する必要がありました。

E2E テストでは、構成ファイルのプロジェクトに定義されているウェブブラウザを起

動し、自動的に page まで用意します。また、テスト終了時にウェブブラウザの終了処理も自動的に行われます。

そのため、ほとんどのケースでいちばん最初に書くコードは、ページの移動になります。ページの移動後は「3.3 Locator」を使い要素の選択や操作をします。Locator は指定した要素が現れるまで自動的に待機するため、要素の出現を待つコードは必要ありません。その後アサーションを行うことにより、テストが実行できます。

また、テストごとに新しいプロファイルのウェブブラウザが作成されるため、他テストの影響を受けません[*1]。そのため、テストごとにテストをしたい状態まで操作する必要があります。

4.1.1 テストの実行

test コマンドでテストが実行できます。

```
npx playwright test
```

特定のファイルを指定したい場合は、続けてパスを指定します。

```
npx playwright test tests/example.spec.js
```

特定のプロジェクトでテストしたい場合は、--project オプションを使います。

```
npx playwright test --project chromium
```

すべてのテストが成功した場合は、次のように表示されます。

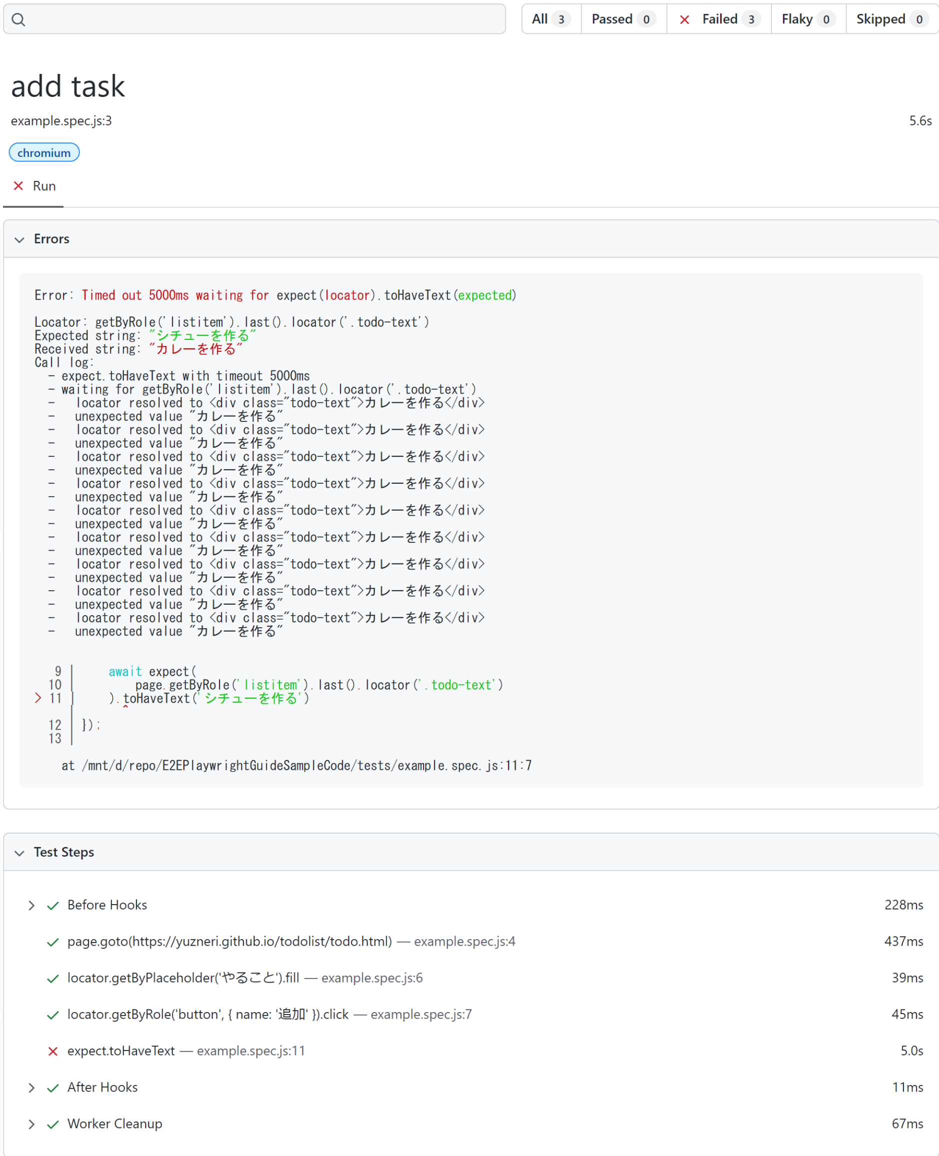
```
Running 3 tests using 3 workers
  3 passed (9.3s)
```

テスト失敗時は、次のように失敗した場所と理由がコンソールに表示されます。

[*1] テストの操作によって、サーバーサイドのデータ更新があるようなケースは除く

```
1) [chromium] ›  example.spec.js:3:1 ›  add task ──────────────────────────────

  Error: Timed out 5000ms waiting for expect(locator).toHaveText(expected)

  Locator: getByRole('listitem').last().locator('.todo-text')
  Expected string: "シチューを作る"
  Received string: "カレーを作る"
  Call log:
    - expect.toHaveText with timeout 5000ms
    - waiting for getByRole('listitem').last().locator('.todo-text')
    -   locator resolved to <div class="todo-text">カレーを作る</div>
    -   unexpected value "カレーを作る"

     9 |     await expect(
    10 |         page.getByRole('listitem').last().locator('.todo-text')
  > 11 |     ).toHaveText('シチューを作る')
       |       ^
    12 | });
    13 |

      at /mnt/d/repo/E2EPlaywrightGuideSampleCode/tests/example.spec.js:11:7

  1 failed
    [chromium] ›  example.spec.js:3:1 ›  add task ──────────────────────────────
```

また、HTML レポートも一緒に表示されます。



▲図 4.1　GitHub のような見た目の HTML レポート

Playwright のあるきかた
〜ゼロから始める E2E テスト〜

2024 年 5 月 25 日　初版第 1 刷　発行

発行所　Neln
印刷所　日光企画