Playwrightで
Visual Regression Testing入門

技育CAMPアカデミア/とっと

自己紹介

- とっと
- XとGitHub：@totto2727
- 株式会社ゆめみ所属
- 入社2年目のフロントエンドエンジニア
- ビジュアルリグレッションテストの経験：半年ぐらい

会社紹介

オープンハンドブック

始める前に

- 以下のページにアクセスし、リポジトリをクローンしておいてください
- https://github.com/totto2727/playwright-vrt-handson

目次

1. なぜテストを書くのか？
2. フロントエンドの色々なテスト（10min）
3. Visual Regression Testing（10min）
4. ハンズオン！基本編（15min）
5. 休憩（10min）
6. 自動テスト(5min)
7. ハンズオン自動テスト編（15min）
8. Next Action（10min）
9. 質問 & ハンズオン応用編

みなさん
テストを書いた経験はありますか？

なぜテストを書くのか？

- コードが正しいこと保証できる
  - サービスを正しく運用できるのか？
  - コードの変更で壊れる部分はないか？
  - 要件を満たすコードになっているか？

フロントエンドの色々なテスト

- 範囲別
  - 静的解析
  - 単体テスト
  - 結合テスト
  - E2Eテスト
- 目的別
  - インタラクションテスト
  - アクセシビリティテスト
  - スナップショットテスト
  - UIコンポーネントエクスプローラ
  - ビジュアルリグレッションテスト
  - etc…

テストの種類が多いよ〜

参考資料：

- 翻訳「フロントエンドアプリケーションの静的、単体、結合、E2Eテスト」 by Kent C. Dodds
- E2Eテストの定義を知りたい

範囲別

静的解析

- コードを実行することなく文字列として解析する
- 型システム：型の整合性を検証し、矛盾を検出する仕組み
  - TypeScript（tsc）
- リンタ：コーディングルールから逸脱したコードを検出する
  - ESLint
  - Biome
- フォーマッタ：コーディングスタイルから逸脱したコードを検出する
  - Prettier
  - Biome

単体テスト

- 関数やコンポーネント単体を対象とするテスト
- 実装、実行ともに低コスト
- 使用されるテストツール：Jest、Vitest
- コンポーネントの評価：jsdomなど

```
// sum.js
export function sum(a, b) {
  return a + b
}

// sum.test.js
import { expect, test } from 'vitest'
import { sum } from './sum'

test('adds 1 + 2 to equal 3', () => {
  expect(sum(1, 2)).toBe(3)
})
```

Vitestの公式サイトから引用

結合テスト

- いくつかの関数、コンポーネントを組み合わせたテスト
- 外部APIなどシステム外に依存する場合はモックを使うことが多い
- 使用されるテストツール：Jest、Vitest
- コンポーネントの評価：jsdomなど

結合テスト

```
import React from 'react'
import {http, HttpResponse} from 'msw'
import {setupServer} from 'msw/node'
import {render, fireEvent, screen} from '@testing-library/react'
import '@testing-library/jest-dom'
import Fetch from '../fetch'

const server = setupServer(
  http.get('/greeting', () => {
    return HttpResponse.json({greeting: 'hello there'})
  }),
)

beforeAll(() => server.listen())
afterEach(() => server.resetHandlers())
afterAll(() => server.close())

test('loads and displays greeting', async () => {
  render(<Fetch url="/greeting" />)

  fireEvent.click(screen.getByText('Load Greeting'))

  await screen.findByRole('heading')

  expect(screen.getByRole('heading')).toHaveTextContent('hello there')
  expect(screen.getByRole('button')).toBeDisabled()
})
```

```
test('handles server error', async () => {
  server.use(
    http.get('/greeting', () => {
      return new HttpResponse(null, {status: 500})
    }),
  )

  render(<Fetch url="/greeting" />)

  fireEvent.click(screen.getByText('Load Greeting'))

  await screen.findByRole('alert')

  expect(screen.getByRole('alert')).toHaveTextContent('Oops, failed to
fetch!')
  expect(screen.getByRole('button')).not.toBeDisabled()
})
```

React Testing Libraryの公式ドキュメントから引用

E2Eテスト

- 本番システム、もしくはそれを模した環境で行う包括的なテスト
- モックなどは基本的に利用しない
- 実装、実行コストは高く、不安定になりやすい
- 使用されるテストツール：Playwright、Cypress、Puppeteer
- コンポーネントの評価：各種ブラウザ

E2Eテスト

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

test('has title', async ({ page }) => {
  await page.goto('https://playwright.dev/');

  // Expect a title "to contain" a substring.
  await expect(page).toHaveTitle(/Playwright/);
});

test('get started link', async ({ page }) => {
  await page.goto('https://playwright.dev/');

  // Click the get started link.
  await page.getByRole('link', { name: 'Get started' }).click();

  // Expects page to have a heading with the name of Installation.
  await expect(page.getByRole('heading', { name: 'Installation' })).toBeVisible();
});
```

Playwrightの公式ドキュメントから引用

目的別

インタラクションテスト

- UIに対して特定のアクションを実行するテスト
  - ボタンのクリック、インプットへの入力 etc…
- 単体、統合テストレベルでは、Jest(Vitest) + React Testing Library
- E2Eテストレベルでは、Playwrightなど

アクセシビリティテスト

- アクセシビリティの要件を満たすか判定するテスト
- 確認項目は色々（デジタル庁のガイドブック）
  - キーボードの操作が可能か？
  - 適切な読み上げが可能か？
  - 色のコントラクト比は十分か？
  - etc…

アクセシビリティテスト

- markuplint
  - HTMLやJSXを静的解析しマークアップが適切か検証できるツール
- Axe
  - 様々なツールと連携できるアクセシビリティテストツール
  - ESLint、Jest、Playwright etc…
- 読み上げツール
  - MacのVoiceOverなど

スナップショットテスト

- 前回のテストと値が一致することを保証するテスト
- フロントエンドの文脈では、コンポーネントをHTML、またはそれを模したオブジェクトや文字列で出力し、比較するテストを指す事が多い
- インタラクションテストと組み合わせることで、特定の操作後のスナップショットを保証するといったことも可能
- 使用されるツール：Jest(Vitest) + React Testing Library etc…

UIコンポーネントエクスプローラ

- コンポーネントやページを一覧するツールおよび手法
- コンポーネントのスタイルや挙動の確認を本実装から切り離せる
- 使用されるツール：Storybook、Ladle
- StorybookにはplayというインタラクションテストをUIの変化を確認しながらテスト、コーディングが可能

ここまで色々紹介してきましたが…

- ロジックに関してはおおよそカバーできる
- しかし、UIに関しては…？
  - スナップショットテストで担保できるのは構造のみ
  - CSSによるスタイリングの変化を検出することは難しい…
  - 様々な場所で利用されるCSS、コンポーネントを修正したときの影響はわからない…
  - UIコンポーネントエクスプローラですべて確認するのも現実的ではない

そこで！

ビジュアルリグレッションテスト

- 以降ではVRTと省略します
- 画像の比較を行い差分を検出するテスト
- ブラウザ上でレンダリングした上でスナップショット（スクリーンショット）を取得し、前回取得した画像と比較する
- 撮影から比較まですべて自動化することで、アプリケーション全体のUIの変更が容易に検知できる

スナップショットテストの比較1

- スナップショットテスト
  - HTMLやそれを模した構造を比較する
  - HTMLの構造や、要素の属性や文言の変化を検出できる
  - CSSの変更は基本検出できない（CSS in JSなどを除く）
- ビジュアルリグレッションテスト
  - HTMLやCSS、JSすべてをブラウザ上でレンダリングし比較する
  - 画像で比較するためCSSの変更による差分も検出できる

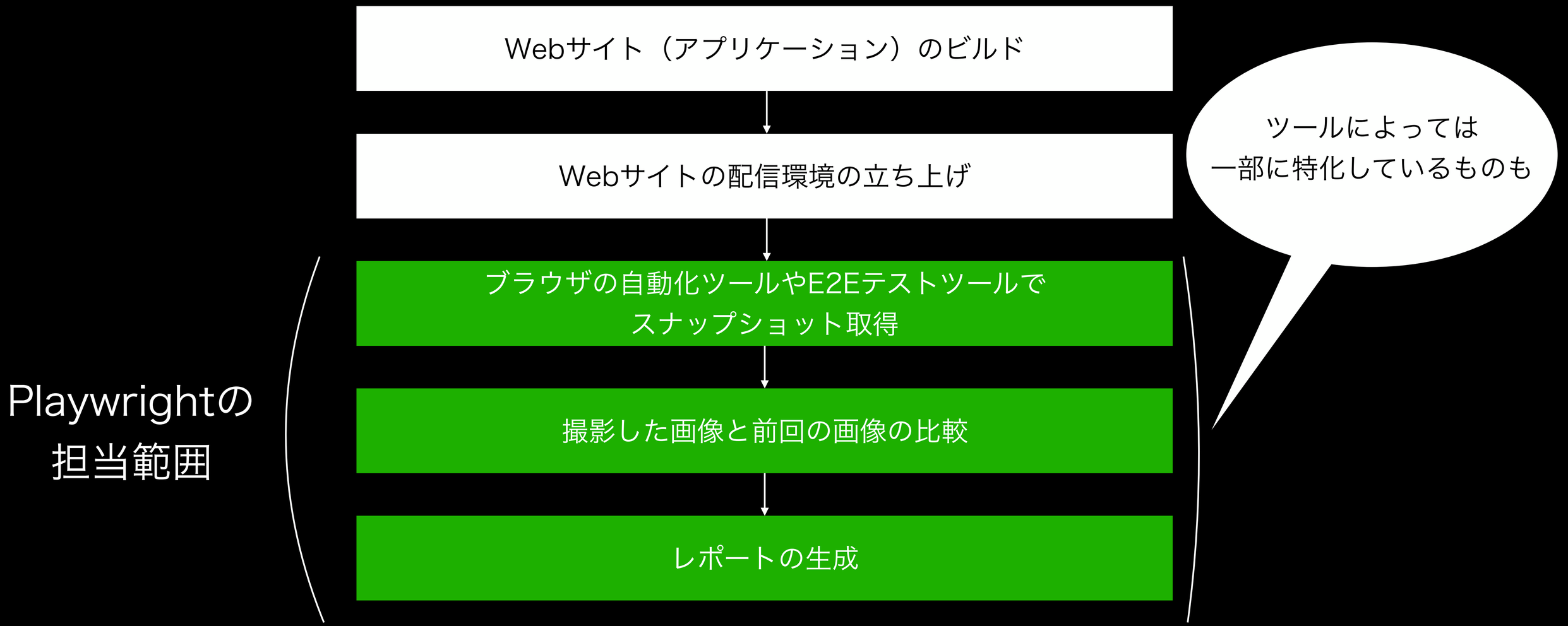
スナップショットテストの比較2

- スナップショットテスト
  - 基本的にJavaScriptの実行のみとなるため実行コストは低い
  - JestやVitestでAPIなどをモックして実行される事が多い
- ビジュアルリグレッションテスト
  - ブラウザを利用するため実行コストが高くなりやすい
  - APIサーバやDBが必要になる可能性もある

VRTの嬉しいところ

- スタイルの変更を検知することができる
  - CSSのリファクタリングを安全に行えるようになる
- 差分の共有が非常に簡単になる
  - 変更前後のスクリーンショットを取得して…といった作業が不要に
  - エンジニア以外のレビューもレポート経由で簡単に

VRTのおおまかな流れ



Playwright

- E2Eテスティングフレームワーク
- ブラウザをコードベースで操作できる
  - JavaScript以外にもPythonなど色々な実装があります
- ボタンのクリックやスクロール、開発者ツールの操作など色々できる
- 様々なブラウザに対応している他、デバイスのプリセットも多くプロジェクトに合わせて柔軟にテストを実現できる

ハンズオン！
～基本編～
目安：15分

CI/CD

- 継続的インテグレーション/継続的デプロイメント
- 今回大事なのはCIの方
- コードの変更などをトリガーに、ビルドやテストを自動で実行し、コードの品質を担保する手法
- Circle CIの解説

GitHub Actions

- CI/CDパイプラインを提供するクラウドサービスの一つ
- GitHubと密接に連携している
  - リポジトリ周りのトリガーが非常に豊富
  - リポジトリがあれば環境構築不要
  - リポジトリの操作が容易
- Publicリポジトリであれば、基本的に無制限に利用可能

GitHub Pages

- GitHubリポジトリと関連するWebサイトをホスティングできるサービス
- 今回はVRTのレポートをホスティングするために利用します
- 1つのドメインしか持てないという欠点はありますが、非常にお手軽に利用できる機能です
  - 同時に複数のブランチからデプロイしたい…！という場合は、NetlifyやVervel、Cloudflare Pagesなどを利用しましょう

ハンズオン！
～自動テスト編～
目安：15分

VRTの利用する際の注意

- ターゲットのサーバが不安定だと、テストの時間が非常に長くかつ不安定になります
  - 可能な限り静的なファイルとして出力して置けるとベスト
- テスト中の待機時間を減らし、並列化することで高速化しましょう
- reg-suit と storycap で行う Visual Regression Testing の高速化

Next Action !

- コンポーネントレベルのVRTを実現する
- テストの設計を行う
- Dockerを利用してCI環境とローカル環境を揃える
- 他のVRTツールを試してみる

コンポーネントレベルのVRT

- フルページのVRTは実装が簡単なように見えて落とし穴が何点かある
  - 大きな差分が出ると本来検出したかった差分が埋もれてしまう
  - Viewport外やアニメーション等など直感的でない場合がある
    - 画像の遅延ロード etc…
- コンポーネントレベルに分割することで、より詳細かつ確実にVRTの恩恵を受けることができる

コンポーネントレベルのVRT

- Playwrightのコンポーネントテスト機能（開発中）
  - 特定のコンポーネントのみブラウザでレンダリングできる
  - 2024/4現在、React、Svelte、Vue、Solidで利用できる
- UIコンポーネントエクスプローラ + VRT
  - Storybookなどと組み合わせることで実現する
  - 代表的な組み合わせ：Storybook + storycap + reg-suit

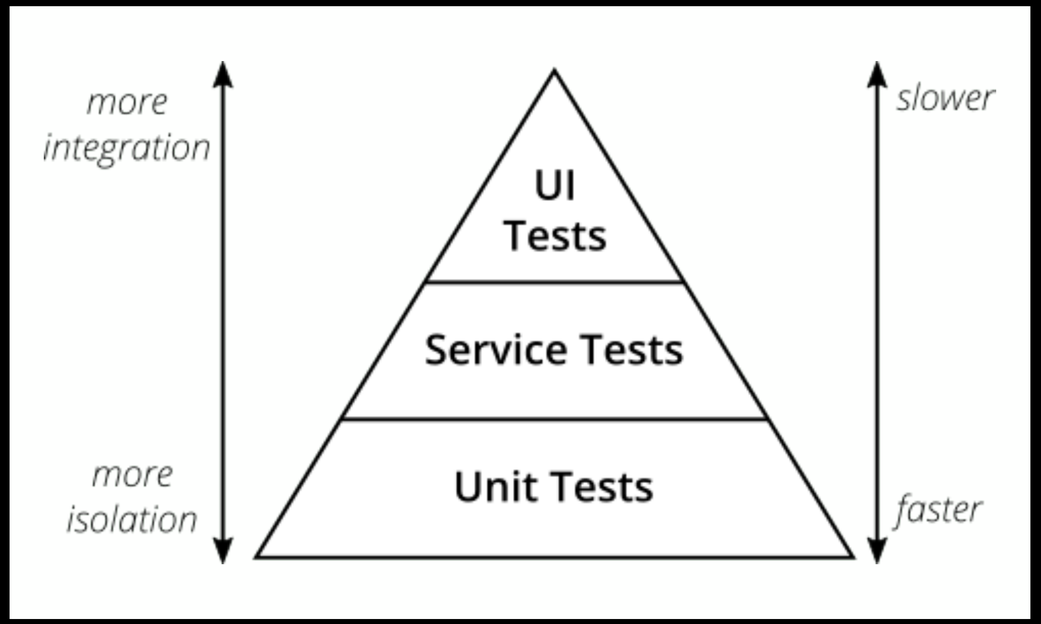
テストの設計

- テスト全体の設計
- VRTの設計

テスト全体の設計

- VRTは便利な手法ですが、テスト全体のバランスを考えることも大切
- 実行コストや何を担保したいのかをよく見極める必要があります
- 代表的なテスト戦略モデル
  - テストピラミッド：実行コストの低いテストを中心にする手法
  - テスティングトロフィー：結合テストを中心とする手法
    - インタラクションテストは統合テストでの実行が中心となるため

テスト全体の設計



The Practical Test Pyramidより引用

The Testing Trophy and Testing Classificationsより引用

VRTの設計

- テストの粒度
- 使用ツール
- スナップショットの更新タイミング
- CI/CDパイプライン
- レポートの共有方法
- レビューフロー

VRTの設計

- テストの粒度
- 使用ツール
- スナップショットの更新タイミング
- CI/CDパイプライン
- レポートの共有方法
- レビューフロー

今回はページ単位

今回はPlaywright

今回はGitHub Actions

今回はGitHub Pages

Dockerを利用したVRT

- 今回のハンズオンでは、ローカルの画像とCIの画像を同期していません
- CIのみで実行する場合は問題にならないですが、ローカルで一部だけ実行したい場合は、フォントなどで差分が出る可能性が高いです
- ローカルとCIの両方でVRTを実行する場合は、同じDocker Imageを利用してテストすることをおすすめします
- 公式ドキュメント

他のVRTツール

- 近年、VRT専門の有用なツール、サービスが登場しています
- クラウドサービス
  - Chromatic、Lost Pixel
- ツール
  - Lost Pixel、reg-suit(とstorycap)
- 2023年にVisual Regression Testingを始めるならどんな選択肢があるか

参考資料の紹介

- フロントエンド開発のためのテスト入門 今からでも知っておきたい自動テスト戦略の必須知識
  - 今必要な基礎知識から実践まで網羅されている
  - ライブラリやツールの仕様変更等もあるため、読むなら今がおすすめ
- ビジュアルリグレッションテストツール４選！ユーザーが語る各ツールのメリット
  - 今回紹介していないPlaywright以外のツールが紹介されています

終わりです！

ありがとうございました！