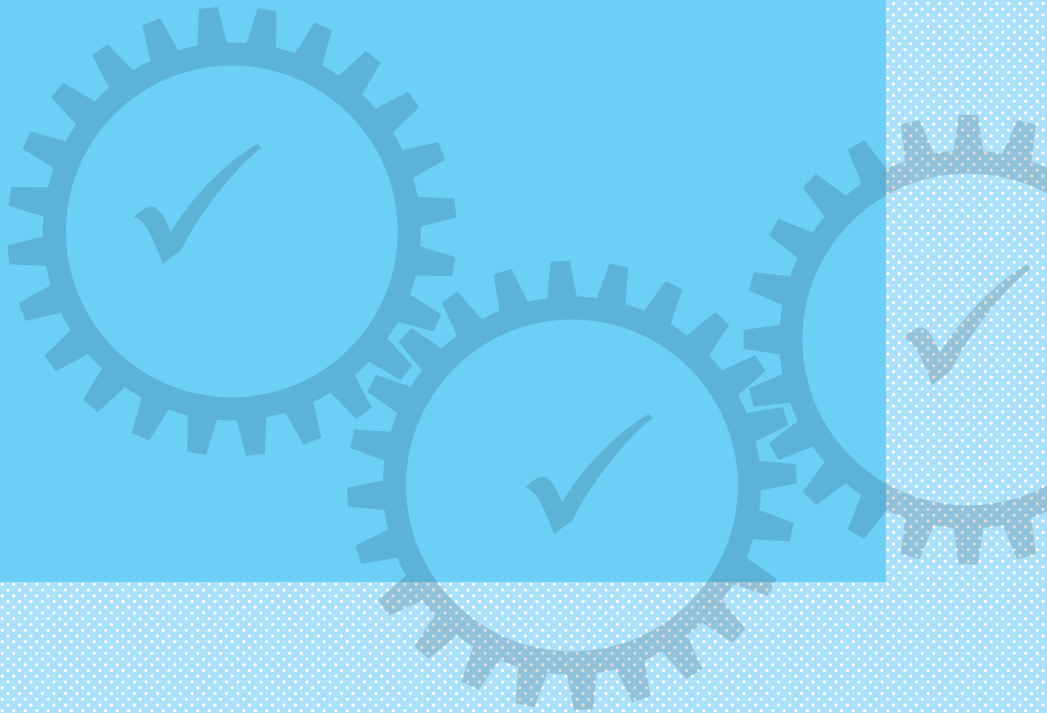
付録A

GitHub Actionsで実行するUIコンポーネントテスト



A-1 GitHub Actionsハンズオン

GitHub Actionsは、ビルド、テスト、デプロイのパイプラインを自動化できるCI/CD（継続的インテグレーション／継続的デリバリー）プラットフォームです。GitHubリポジトリにyamlファイルを置くだけで、自動テストをすぐにはじめることができます。概要を理解するため、まずはリポジトリを作り一番簡単なテストをGitHub Actionsで動かしてみましょう。

● リポジトリを用意する

名称は何でもよいですが、本書では`github-actions-example`という名称でリポジトリを作成します。Node.jsがインストールされている開発環境にリポジトリをクローンしたら、`npm init`を実行し、`package.json`を作成します。そして、次のコマンドでJestをインストールします。

```bash
$ npm install jest --save-dev
```

そして次に示すのが`package.json`に必要な最小構成です（リストA-1）。

▶ リストA-1　package.json

```json
{
  "name": "github-actions-example",
  "private": true,
  "scripts": {
    "test": "jest"
  },
  "devDependencies": {
    "jest": "^29.2.1"
  }
}
```

●最も単純なテストを書く

最も単純なテストとして、Jestの公式サイトで紹介されている関数とテストを用意します（リストA-2、リストA-3）。

▶リストA-2　sum.js

```JavaScript
function sum(a, b) {
  return a + b;
}
module.exports = sum;
```

▶リストA-3　sum.test.js

```JavaScript
const sum = require("./sum");

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

この状態で、npm testを実行します。テストが実行されていることが確認できます。

```bash
 PASS  ./sum.test.js
  ✓ adds 1 + 2 to equal 3 (1 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.306 s
```

●ワークフローファイルを作成する

次に、ワークフローファイルを作成します（リストA-4）。GitHub Actionsのワークフローとは、一連のジョブ実行を構成する自動化プロセスのことです。リポジトリに.github/workflowsというディレクトリを作成し、ワークフローファイル（yaml）を置きます。これだけで、GitHub Actionsが実行されるようになります。

▶リストA-4　.github/workflows/ci.yaml

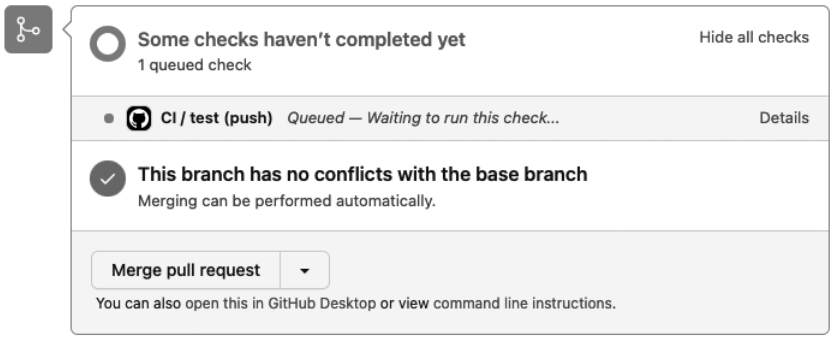
```yaml
name: CI

on: push                                    # リポジトリにpushする度に実行する

jobs:
  test:
    runs-on: ubuntu-latest                  # OSは最新のUbuntuを使用する
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3         # CI環境でNode.jsを使用する
        with:
          node-version: 18
      - name: Install dependencies          # CI環境にpackage.json内容をインストール
        run: npm ci
      - name: Jest unit test #CI環境でnpm testを実行
        run: npm test                       # テストを実行する
```

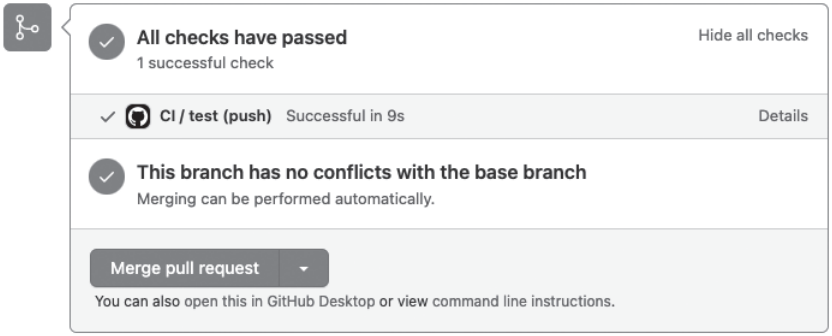
●プルリクエストを作成する

ブランチをリモートリポジトリにpushし、プルリクエストを作成してみましょう。アイコンが黄色のときは、テストが進行中であることを表しています（図A-1）。



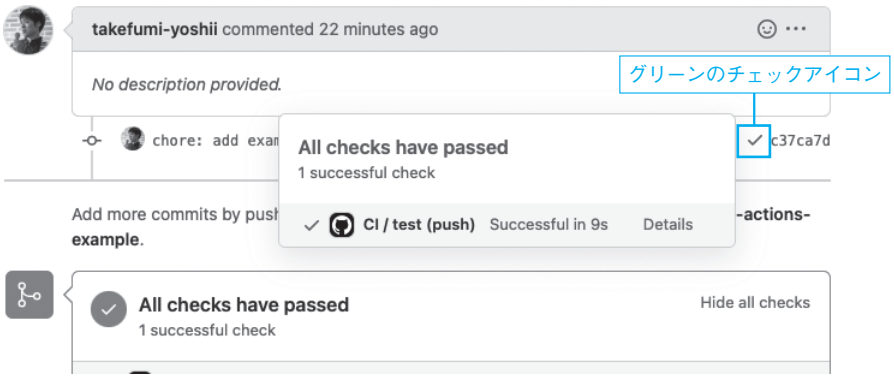
図A-1　テストが進行中のプルリクエストのメッセージ

1分ほど待つとアイコンがグリーンになり、テストが成功したことが通知されます（図A-2）。



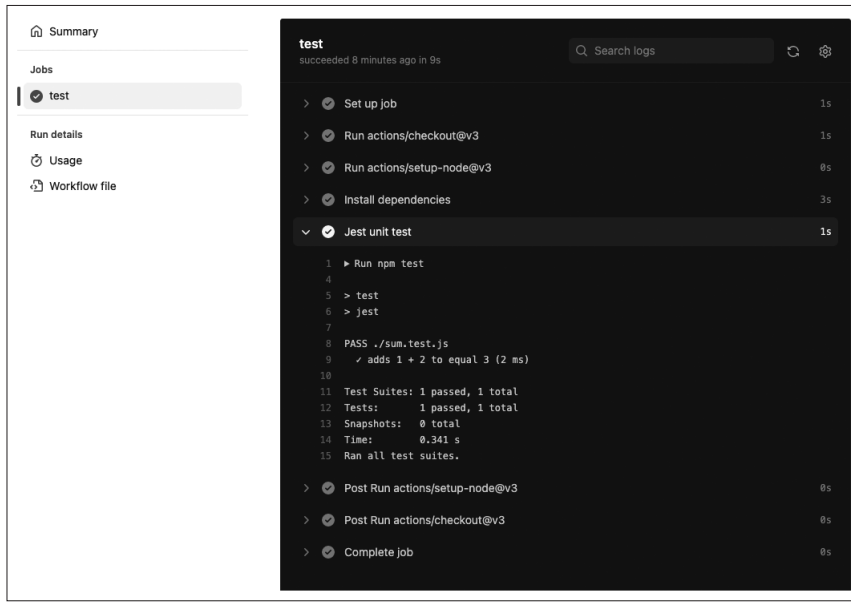
図A-2　テストが成功したプルリクエストのメッセージ

コミットハッシュの横には、グリーンのチェックアイコンが表示されています。これをクリックするとポップアップが表示されます（図A-3）。



図A-3　テストが成功したコミットのポップアップ

このポップアップに含まれる「Details」と書かれたテキストリンクをクリックすると、実施されたジョブの詳細内訳画面に遷移します（図A-4）。



図A-4　実施されたジョブの詳細内訳画面（全てパスしている）

テストステップにあたる「Jest unit test」ノードをクリックすると、開発環境で実行した内容と同じログが出ていることが確認できます。この詳細画面を確認することで、どういった経緯でテストがパスしたか、あるいはしなかったのかを、検証できます。

●テストが失敗することを試す

テストが失敗するとどのようになるか見てみましょう。先ほどのテストを失敗するように改変します（リストA-5）。

▶リストA-5　sum.test.js

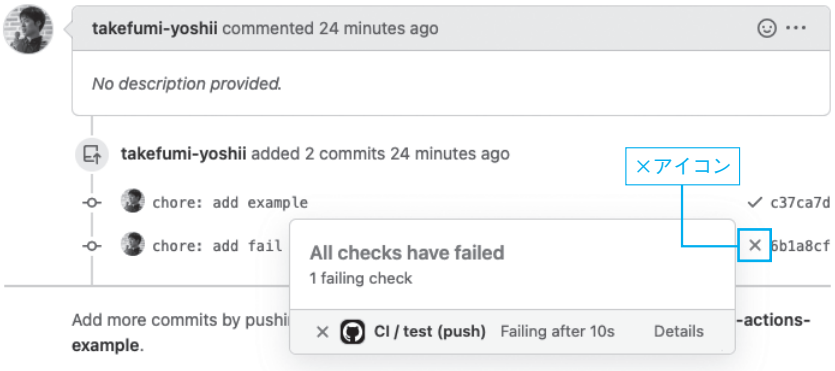
JavaScript

```
const sum = require("./sum");

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

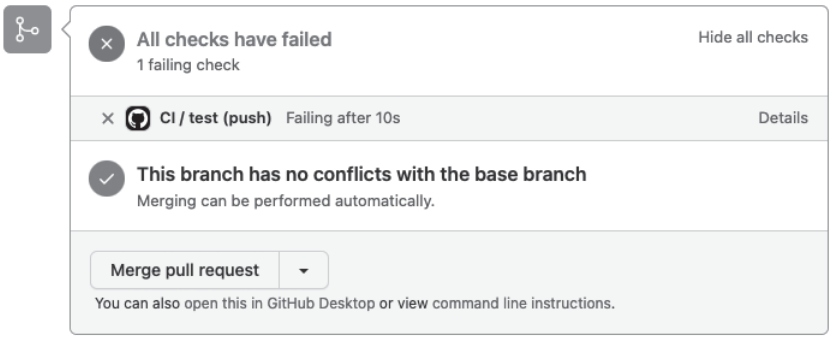
2 + 2に変更

プルリクエストにコミットを追加すると、再度テストが実行されます。結果は意図通りに失敗し、コミットハッシュの横に「×」アイコンが表示されていることが確認できます（図A-5）。



図A-5　テストが失敗したコミットのポップアップ

プルリクエスト下部のメッセージからも、失敗している旨が確認できます（図A-6）。



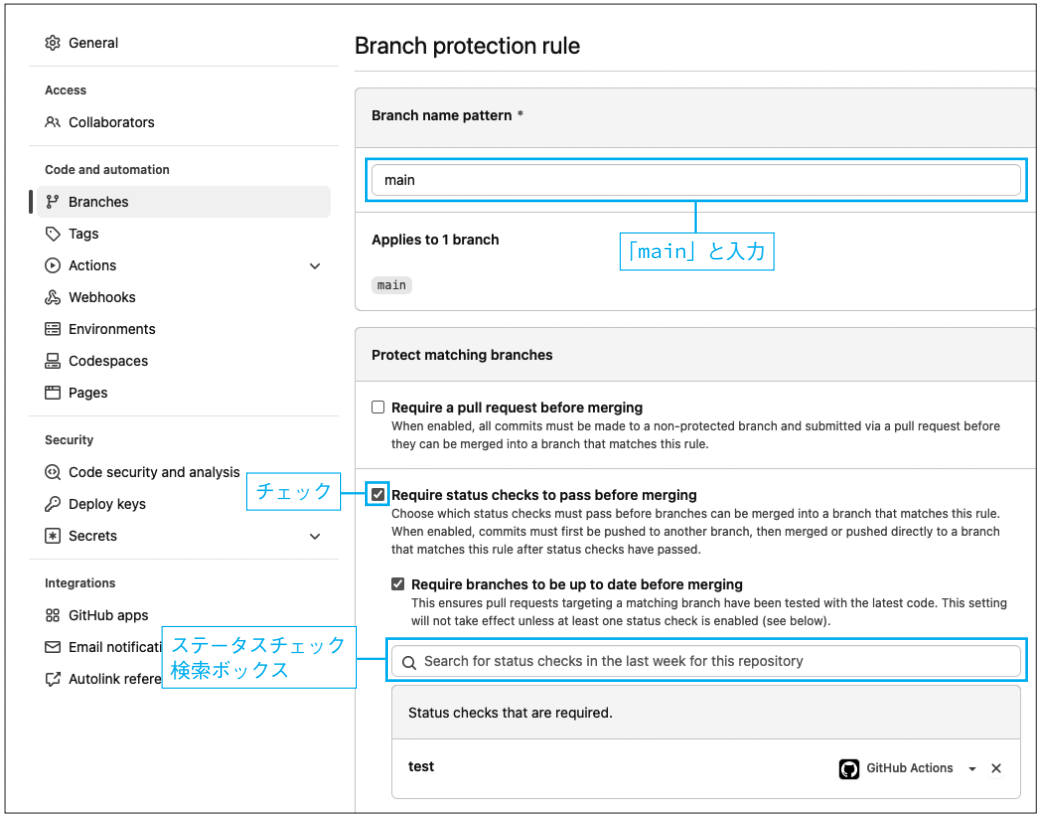
図A-6　テストが失敗したプルリクエストのメッセージ

● テストが失敗したプルリクエストをマージできないようにする

新規リポジトリ作成直後では、テストが失敗した状態でもマージができてしまいます。うっかりマージしてしまわないよう、リポジトリの「Settings -> Branches」ページで、ブランチ保護ルール（Branch protection rule）を設定してみましょう。

まず、ルールを適用したいブランチ名のパターンを`Branch name pattern`に入力します。`main`ブランチに対するプルリクエストには全て適用するものとして、`main`と入力します。

続いて、`Require status checks to pass before merging`（マージ前にチェックステータスがパスする必要がある）にチェックを入れます（図A-7）。

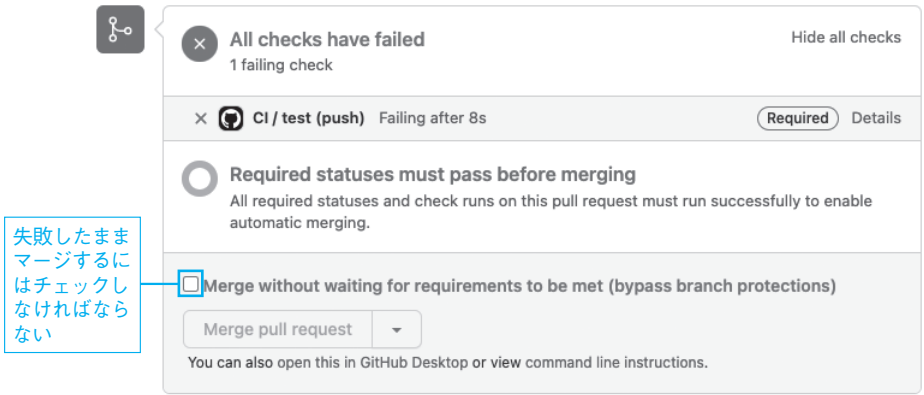


図A-7　ブランチ保護ルールの設定画面

そして、下のステータスチェック検索ボックスにジョブ名称を入力すると、該当ジョブによるステータスチェックが適用対象として選択できます。`test`（yamlファイルに指定したジョブ名称）を入力し、プルダウン表示一覧から`test`を選択しましょう。

最後に、ページ下部の「Save changes」ボタンを押下すれば設定完了です。テストが失敗したプルリクエストの画面を確認すると、先ほどとは異なるメッセージ表示になっていることが確認できます（図A-8）。

失敗した状態でマージするにはチェックボックスにチェックを入れる必要があるので、うっかりマージしてしまうということもなくなりそうです。このメッセージ表示はどこかに不具合が含まれていることを意味するので、プルリクエストの内容を見直すようにしましょう。



図A-8　テストが失敗したプルリクエストのメッセージ

A-2 ワークフローファイルの書き方

GitHub Actionsのワークフローとは、一連のジョブ実行を構成する自動化プロセスのことです。ワークフローをスタートさせる「トリガーの定義」、ワークフロー内訳を記す「ジョブの定義」を組み合わせ、任意のCI/CDパイプラインを構成できます。記述方法は多岐にわたるため、本書で紹介しているサンプルコードに含まれるものを中心に解説していきます。

● 名称（name）

ワークフローファイルのトップレベル（ルートノード）で定義されるnameは、ワークフローを特定するための名称です。実行結果などを識別しやすいよう、わかりやすい名称を与えます。この名称は、プルリクエストやActions履歴などにラベルとして使用されます。

yaml
```
name: Test UI
```

● 環境変数（env）

ワークフローファイルのトップレベルで定義されるenvは、ワークフロー内のジョブから参照可能な環境変数です。

yaml
```
env:
  REG_NOTIFY_CLIENT_ID: abcd1234
  AWS_BUCKET_NAME: my-bucket-name
```

環境変数は、パスワードやIDなど、クレデンシャル情報を扱うときに参照されることが多いです。こういった情報はソースコードにコミットしてはならず、適切な権限下で管理される必要があります。リポジトリごとの「Actions secrets」で環境変数を事前に設定しておくと、ワークフローファイルから参照できるようになります。ワークフローから、クレデンシャル情報を参照する書き方が次のものです。ワークフローファイルの${{…}}は文字列代入のテンプレート構文であり`secrets.`で、「Actions secrets」が参照できます。

yaml
```
env:
  REG_NOTIFY_CLIENT_ID: ${{ secrets.REG_NOTIFY_CLIENT_ID }}
  AWS_BUCKET_NAME: ${{ secrets.AWS_BUCKET_NAME }}
```

● トリガーの定義（on）

ワークフローファイルのトップレベルで定義されるonは、ワークフローをスタートさせるタイミングを指定します。例えば、プッシュする度にテストが実行されてほしい場合、次のように指定します。

```yaml
on: push
```

この指定は、リポジトリがモノレポ構成※A-1の場合に問題が発生します。それは、テスト対象とは関係ないファイルがプッシュされたときにも、関係のないワークフローがスタートしてしまうという問題です。「特定のpackageに含まれるソースコードがプッシュされたとき」という指定は、次のように記述します。これで、packages/app向けのワークフローファイルとすることができます。

```yaml
on:
  push:
    paths:
      - packages/app/**
```

● ジョブの定義（jobs）

ワークフローファイルのトップレベルで定義されるjobsは、ワークフローをスタートさせるタイミングを指定します。

runs-on

ジョブが実行される仮想マシンのOSを指定します。サンプルコードではubuntu-latest（最新のUbuntu）にしていますが、必要に応じて任意のOSを選択できます。選択できるOSは公式ドキュメントを参照してください。

```yaml
runs-on: ubuntu-latest
```

steps

ジョブで実行される全てのステップをグループ化します。このグループに個別のアクションまたはシェルスクリプトを含めます。

```yaml
steps:
```

※A-1 1つのリポジトリで、バックエンド／フロントエンドなど、複数領域のコードを管理する構成のことです。

uses

usesキーワードは、ステップでアクションを使用することを宣言します。最も使用頻度の高いアクションはactions/checkoutです。actions/checkoutは、リポジトリをランナーにチェックアウトするアクションなので、ほとんどのジョブで必要になります。

yaml
```
- uses: actions/checkout@v3
```

フロントエンドプロジェクトでは、actions/setup-nodeもほとんどのジョブで必要になります。次のように任意バージョンのNode.jsを指定して、実行環境をインストールします。

yaml
```
- uses: actions/setup-node@v3
  with:
    node-version: 18
```

run

runキーワードは、ランナーでコマンドを実行するようにジョブに指示します。プロジェクトで実行可能なコマンドが指定できるため、package.jsonに記載したnpm scriptsも実行できます。

yaml
```
run: npm test
```

本書ではCI（継続的インテグレーション）のみを紹介していますが、アクションを組み合わせたり、シェルスクリプトを登録したりすることで、CD（継続的デリバリー）のパイプラインも構築できます。ビルド済みのアプリケーションを配布するだけでなく、ビルド済みのStorybookをホスティング環境に転送するといったこともできるため、フロントエンド開発環境構築に活用していきましょう。

A-3 jobを並列実行してワークフローを高速化する

ジョブを実行した結果の成果物（インストールしたモジュールやビルドしたアセットなど）は、キャッシュすることができます。このキャッシュを活かすことで、ワークフローの実行時間が短縮されます。また、キャッシュ経由で、複数ジョブ間で成果物を共有することができます。

ここまでのサンプルでは単一ジョブを実行するのみでしたが、GitHub Actionsのワークフローは複数のジョブを並列実行することができます。うまく組み合わせることで、全てのジョブが完了するまでの待ち時間を短縮することができます。

●単一ジョブのワークフロー

第7章～第10章で使用したNext.jsサンプルリポジトリには、UIコンポーネントテストが数種類含まれています。これらのテストを単一ジョブで実行する場合、次のワークフローファイルで全て実行することができます（リストA-6）。

▶リストA-6　単一ジョブで全てのテストを実行する例

yaml

```yaml
name: Test UI

on: push

env:
  REG_NOTIFY_CLIENT_ID: ${{ secrets.REG_NOTIFY_CLIENT_ID }}
  AWS_BUCKET_NAME: ${{ secrets.AWS_BUCKET_NAME }}

jobs:
  tests:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
        with:
          fetch-depth: 0
      - uses: actions/setup-node@v3
        with:
          node-version: 18
      - name: Configure AWS Credentials
```

```
      uses: aws-actions/configure-aws-credentials@master
      with:
        aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
        aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
        aws-region: ap-northeast-1
    - name: Install dependencies
      run: npm ci
    - name: Run Type Check
      run: npm run typecheck
    - name: Run Lint
      run: npm run lint
    - name: Run Unit Tests
      run: npm test
    - name: Build Storybook
      run: npm run storybook:build --quiet
    - name: Run Storybook Tests
      run: npm run storybook:ci
    - name: Run Storycap
      run: npm run vrt:snapshot
    - name: Run reg-suit
      run: npm run vrt:run
```

依存modulesのインストール

型チェック実行

Lint実行

単体テスト実行

Storybookのビルド

StorybookのUIテスト

Storybookのキャプチャ

Storybookのビジュアルリグレッションテスト

ワークフローを実行した結果を、リポジトリの「Actions」タブから確認してみましょう。結果の概要ページにいくと「tests」というジョブに約7分かかっていることが確認できます（図A-9）。待ち時間を節約するため、このような単一ジョブを細分化し、並列実行するように組み替えて、高速化に挑戦していきます。

図A-9　単一ジョブのワークフロー

● jobの並列実行を計画する

開発環境と同じように、CI環境でnode_modulesをインストールすると、次のタスクが実行できるようになります。この3つのタスクには依存関係がないため、各ジョブに分割して並列実行できそうです。

- 型チェック、Lint実行
- 単体テスト実行
- Storybookのビルド

次の2つのタスクはStorybookのビルドを待ち、その成果物を対象にテストを行います。UIテスト／ビジュアルリグレッションテストは相互の依存はないため、並列実行できそうです。

- StorybookのUIテスト
- Storybookのビジュアルリグレッションテスト

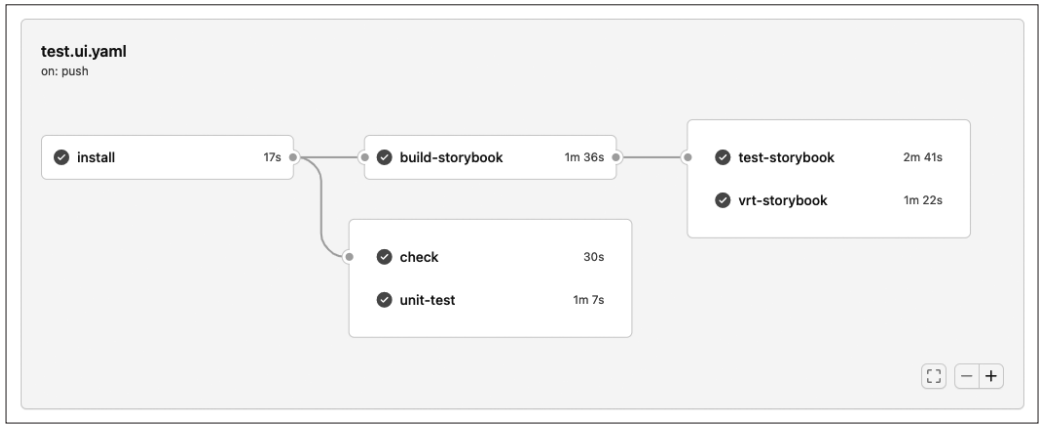
整理すると、次のように①〜③ステップに区分できます（リストA-7）。そして各々のステップでは、並列実行ができそうです。

▶リストA-7　jobの並列実行順

yaml

```
jobs:
  install: …          ←  ①依存モジュールのインストール
  check: …            ←  ②型チェック、Lint
  unit-test: …        ←  ②単体テスト
  build-storybook: …  ←  ②Storybookビルド
  test-storybook: …   ←  ③StorybookのUIテスト
  vrt-storybook: …    ←  ③Storybookのビジュアルリグレッションテスト
```

この依存関係の通りにジョブを組み替えたワークフローを実行すると、次のキャプチャのような実行結果が得られます（図A-10）。同じタイミングで実行されたジョブが縦軸で並び、依存関係が線で連結されている様子が、グラフで確認できます。そして、全てのジョブが完了するまでの時間が約5分に短縮されていることがわかります。



図A-10　複数ジョブのワークフロー

● jobの依存関係を構成する

紹介したワークフローを構成するため、ジョブ間の依存関係を設定します（リストA-8）。ジョブの「needs」プロパティに、依存しているジョブ名称を指定すると、依存先のジョブが完了するまで該当ジョブは待機します。これでジョブ間の依存関係が指定できました。

▶ リストA-8　jobの依存関係を構成する

yaml

```
jobs:
  install: …                                    ①
  check:
    needs: install                              ①のinstallが完了するのを待つ
    steps: …
  unit-test:
    needs: install                              ①のinstallが完了するのを待つ
    steps: …
  build-storybook: …                            ②
    needs: install                              ①のinstallが完了するのを待つ
    steps: …
  test-storybook:
    needs: build-storybook                      ②のbuild-storybookが完了するのを待つ
    steps: …
  vrt-storybook:
    needs: build-storybook                      ②のbuild-storybookが完了するのを待つ
    steps: …
```

●依存のあるジョブ間でキャッシュを利用する

全てのジョブは、プロジェクトにインストールするnode_modulesが必要になります。はじめにnpm ciでnode_modulesをインストールします。しかし、インストールしたnode_modulesは、そのままでは後続のジョブで共有することができません。

GitHub Actions公式のactions/cacheを使用すると、指定した名称でアセットをキャッシュしたり、キャッシュからアセットを復元することができます。あるジョブでキャッシュしたアセットを別のジョブで復元することで、独立した複数のジョブ間におけるアセット共有が実現できます。

はじめに「install」ジョブを確認していきましょう。if文に記載されている条件は「node_modules_cacheというIDで特定できるステップのキャッシュヒットがなければ」という内容です。つまり、もしキャッシュヒットしなければインストールし、キャッシュヒットすればステップを終了するというものです。

「path」に記載されているのはキャッシュ対象へのパス、「key」はキャッシュに使用するキーです。この例では、hashFilesという関数を使い、package-lock.jsonの内訳からハッシュキーを生成しています。この指定をすることで「package-lock.jsonに更新がなければキャッシュを再利用する」ことが実現できます（リストA-9）。

▶リストA-9　依存のあるジョブ間でキャッシュを利用する

yaml

```yaml
jobs:
  install:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: 18
      - name: Cache node_modules
        uses: actions/cache@v3
        id: node_modules_cache
        with:
          path: node_modules
          key: ${{ runner.os }}-${{ hashFiles('**/package-lock.json') }}
      - name: Install dependencies
        if: steps.node_modules_cache.outputs.cache-hit != 'true'
        run: npm ci
```

キャッシュヒットするか検証する

キャッシュ対象の依存modules

キャッシュヒットしなければインストールする

● キャッシュしたnode_modulesを復元する

「check」と名付けたジョブでは、型チェック、Lintを実行します（リストA-10）。「Restore node_modules」というステップでは「install」でキャッシュしたnode_modulesを復元しています。キャッシュ時と同じ指定になりますが、このステップを挟むことで、ワーキングディレクトリにキャッシュしたアセットが復元されます。

同時に並列実行する「unit-test」ジョブも、同じ要領でキャッシュを復元してからテストを実行します。

▶ リストA-10　キャッシュしたnode_modulesを復元する

yaml

```
jobs:
  install: …
  check:
    needs: install
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: 18
      - name: Restore node_modules
        uses: actions/cache@v3
        id: node_modules_cache
        with:
          path: node_modules
          key: ${{ runner.os }}-${{ hashFiles('**/package-lock.json') }}
      - name: Run Type Check
        run: npm run typecheck
      - name: Run Lint
        run: npm run lint
```

キャッシュから依存modulesを復元する

型チェック実行

Lint実行

● ビルドしたStorybookをキャッシュする

Storybookを使用したテストを安定して実行するため、CIでは事前にStorybookのビルドを行います。node_modulesをインストールしたときと同じように、ビルド済みのStorybookを後続のジョブに向けてキャッシュします。今回はキャッシュキーにgitHub.shaを含めているため、同じコミットでのみ共有可能なキャッシュということになります（リストA-11）。

▶ リストA-11　ビルドしたStorybookをキャッシュする

yaml

```yaml
jobs:
  install: …
  check: …
  unit-test: …
  build-storybook:
    needs: install
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: 18
      - name: Restore node_modules
        id: node_modules_cache
        uses: actions/cache@v3
        with:
          path: node_modules
          key: ${{ runner.os }}-${{ hashFiles('**/package-lock.json') }}
      - name: Cache Storybook
        uses: actions/cache@v3
        id: storybook_cache
        with:
          path: storybook-static
          key: ${{ runner.os }}-${{ github.sha }}
      - name: Build Storybook
        if: steps.storybook_cache.outputs.cache-hit != 'true'
        run: npm run storybook:build --quiet
```

キャッシュから依存modulesを復元する

ビルド済みStorybookキャッシュの準備

キャッシュからビルド成果物を復元する

● ビルド済みのStorybookを復元、テスト対象として使用する

最後に、ビルド済みのStorybookを復元してテストを実行します（リストA-12）。UIテスト／ビジュアルリグレッションテストは、このビルド済みのStorybookを使用します。

▶ リストA-12　ビルド済みのStorybookを復元、テスト対象として使用する

yaml

```yaml
jobs:
  install: …
  check: …
  unit-test: …
  build-storybook: …
  test-storybook:
    needs: build-storybook
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: 18
      - name: Restore node_modules
        id: node_modules_cache
        uses: actions/cache@v3
        with:
          path: node_modules
          key: ${{ runner.os }}-${{ hashFiles('**/package-lock.json') }}
      - name: Restore Storybook
        id: storybook_cache
        uses: actions/cache@v3
        with:
          path: storybook-static
          key: ${{ runner.os }}-${{ github.sha }}
      - name: Install Playwright
        run: npx playwright install --with-deps chromium
      - name: Run Storybook Tests
        run: npm run storybook:ci
```

キャッシュから依存modulesを復元する

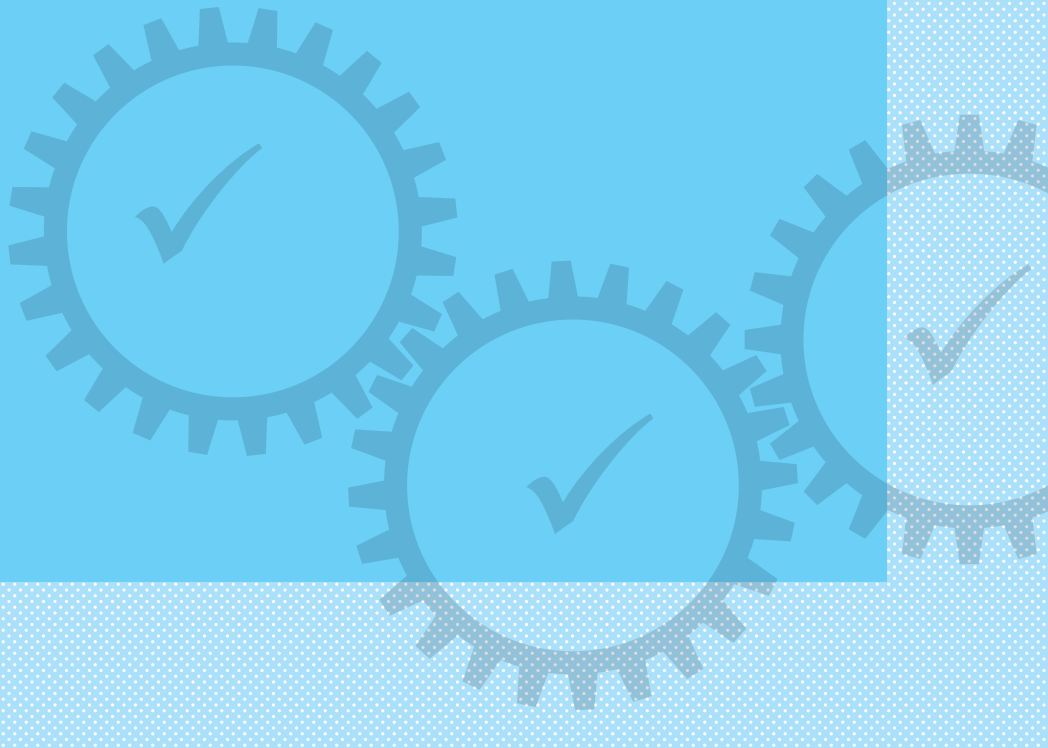
キャッシュからビルド済みStorybookを復元する

Storybook UIテストに必要な依存modules

ジョブを組み替えたことにより、実行時間の短縮が実現できました。テストコードが増えたとき、ジョブの並列実行はより効果が表れるでしょう。また、package-lock.jsonはプッシュやプルリクエストの度に変わるものではなく、単一ジョブのワークフローでも活きる指定なので活用していきましょう。

付録 B

GitHub Actionsで実行するE2Eテスト



B-1 GitHub Actionsで E2Eテストを実行する

DBや外部サーバー連携があるアプリケーション開発に、Docker Composeを使用することが多くなりました。DBや外部サーバー連携があるE2Eテストも同様に、Docker Composeを使用することでシステム構成の再現が可能です。

第10章で解説したサンプルでは、開発環境で実行するE2Eテストを紹介しました。このサンプルには本章で使用するGitHub Actions E2Eテストのための「Dockerfile & Docker Composeファイル」も含まれています。

それは、複数あるコンテナーのうちの1つに「E2Eテストを実行、終了する」というコンテナーを含めるものです。実行したプロセスが0のシグナルをもって終了した場合、ジョブは成功と判定されます。一方で、1をもってexitした場合、ジョブは失敗と判定されます。

Docker Composeで関連するコンテナー群を起動し、E2Eテストを実行するコンテナーを最後に起動することで、仮想マシンの中でE2Eテストが完結します。これが本章で紹介する、GitHub Actionsで実行するE2Eテストの概要です。

● ワークフローファイルの内訳

サンプルコード「.github/workflows/test.e2e.yaml」のE2Eテスト向けワークフローファイルは、次のようになっています（リストB-1）。

▶ リストB-1　.github/workflows/test.e2e.yaml

```yaml
name: Test E2E

on: push

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: 18
      - name: Install dependencies
        run: npm ci
```

```
      - name: Docker Compose Build
        run: npm run docker:e2e:build
      - name: Docker Compose Up As E2E Testing
        run: npm run docker:e2e:ci
```

- Install dependencies：node_modulesのインストール
- Docker Compose Build：E2Eテスト向けのビルド
- Docker Compose Up As E2E Testing：E2Eテストの実行

● npm scriptsの内訳

リストB-2は、ワークフローに記載されているnpm scriptです。`-f docker-compose.e2e.yaml`オプションでファイル名を指定し、ビルドと起動を行っています。`npm run docker:e2e:ci`が起動コマンドになりますが、着目すべき指定は`--exit-code-from e2e`です。これは「e2eという名前のコンテナー終了シグナルをもってDocker Composeを終了する」という指定です。この結果次第で、ジョブの成否（E2Eテストが成功したか否か）が決まります。

▶リストB-2　package.json

```json
{
  "scripts": {
    "docker:e2e:build": "docker compose -f docker-compose.e2e.yaml build",
    "docker:e2e:ci": "docker compose -f docker-compose.e2e.yaml up ➡
--exit-code-from e2e"
  }
}
```

B-2 Dockerfileを作成する

GitHub Actions1つ目のステップ「Docker Compose Build」を見ていきましょう。ここでは、docker-composeのビルドを行っています（リストB-3）。

▶リストB-3　package.json

```json
{
  "scripts": {
    "docker:e2e:build": "docker compose -f docker-compose.e2e.yaml build"
  }
}
```

対象の「docker-compose.e2e.yaml」ファイルのビルド設定のうち、E2Eテストのためのイメージビルドに「Dockerfile.e2e」が指定されています（リストB-4）。

▶リストB-4　docker-compose.e2e.yaml

```yaml
services:
  db: …
  redis: …
  minio: …
  createbuckets: …
  e2e:
    build:
      context: .
      dockerfile: Dockerfile.e2e
      args:
        DATABASE_URL: postgresql://root:password@db:5432/app-db?schema=public
```

このDockerfileではマルチステージビルドで構成されており、次の3つのステージで構成されています。

- deps：node_modulesのインストール
- builder：Next.jsアプリケーションをビルド
- runner：アプリケーション起動とE2Eテスト実行

●depsステージの内訳

依存モジュールをインストールするステージです。package.jsonとpackage-lock.jsonをコピーして、node_modulesをインストールします（リストB-5）。

▶リストB-5　Dockerfile.e2e

Dockerfile

```
# deps
FROM node:18-alpine AS deps
RUN apk add --no-cache libc6-compat

WORKDIR /app

COPY package.json package-lock.json ./

RUN npm ci
```

●builderステージの内訳

アプリケーションをビルドするステージです。depsでインストールしたnode_modulesをステージにコピーします。npm run buildを実行するとNext.jsアプリケーションビルドが実行されますが、その前にnpx prisma generateが実行されています。

これはNext.jsアプリケーションで使用しているORMのPrismaでSchemaファイルを読み取り、Prisma Clientを先に作成する必要があるためです。環境変数のDATABASE_URLはPrismaに向けたものです（リストB-6）。

▶リストB-6　Dockerfile.e2e

Dockerfile

```
# builder
FROM node:18 AS builder

WORKDIR /app

COPY --from=deps /app/node_modules ./node_modules
COPY . .

ARG DATABASE_URL
ENV DATABASE_URL=$DATABASE_URL
ENV NEXT_TELEMETRY_DISABLED 1

RUN npx prisma generate
RUN npm run build
```

NEXT_TELEMETRY_DISABLEDは、Next.jsのビルド時に送信されるテレメトリーを無効にするオプションです。詳しくは以下のページをご確認ください。

URL　https://nextjs.org/telemetry

● runnerステージの内訳

Docker Hubで配布されている、Microsoft公式の「Playwright」イメージを使用します。builderステージから次のものをコピーします（リストB-7）。

- ビルドしたNext.jsアプリケーション
- Prisma実装（E2Eテスト実施直前のマイグレーション用）
- E2Eテストファイル、Playwrightコンフィグ

▶リストB-7　Dockerfile.e2e

Dockerfile

```
# runner
FROM mcr.microsoft.com/playwright:v1.27.1-focal AS runner

WORKDIR /app

COPY --from=builder /app/package.json           package.json
COPY --from=builder /app/public                 public
COPY --from=builder /app/.next                  .next
COPY --from=builder /app/prisma                 ./prisma
COPY --from=builder /app/e2e                    e2e
COPY --from=builder /app/playwright.config.ts   playwright.config.ts
COPY --from=builder /app/node_modules           node_modules

EXPOSE 3000

ENV NEXT_TELEMETRY_DISABLED 1
ENV NODE_ENV production
ENV CI true
ENV PORT 3000

CMD ["npm", "run", "docker:e2e:start"]
```

● npm scriptsの内訳

このコンテナーを起動すると`docker:e2e:start`というnpm scriptが実行されます（リストB-8）。このコマンド1つで、次の3つが順番に実行されます。

- `npm run prisma:reset`で、DBの初期データを投入
- `npm start`で、Next.jsアプリケーションを起動
- `npm run test:e2e`で、E2Eテスト実行

▶リストB-8　package.json

```json
{
  "scripts": {
    "start": "next start",
    "prisma:reset": "prisma migrate reset --force",
    "test:e2e": "npx playwright test",
    "docker:e2e:start": "npm run prisma:reset && start-server-and-test ➡
'npm start' 3000 'npm run test:e2e'",
  }
}
```

start-server-and-testというnpm packageは、アプリケーションサーバー起動とテスト実行が同時に行えるものです。アプリケーションサーバーの起動を待ってから、テストを実施します。そしてテストが完了したら、アプリケーションサーバーを終了します。

B-3 Docker Composeファイルを作成する

GitHub Actions2つ目のステップ「Docker Compose Up As E2E Testing」を見ていきましょう。ここでは、docker-compose E2Eテストを実施します（リストB-9）。

▶リストB-9　package.json

```json
{
  "scripts": {
    "docker:e2e:ci": "docker compose -f docker-compose.e2e.yaml up ➡
--exit-code-from e2e"
  }
}
```

● コンテナーの依存関係

はじめに、E2Eテストに必要なコンテナー群の依存関係をdepends_onで構成します（リストB-10）。depends_onが設定されているコンテナーは、依存先のコンテナー起動が完了した後に起動するという指定です。e2eコンテナーに含まれるcreatebucketsの起動を待っている点について、確認していきましょう。

▶ リストB-10　docker-compose.e2e.yaml

```yaml
services:
  db: …
  redis: …
  minio: …
  createbuckets:
    depends_on:
      - minio
  e2e:
    depends_on:
      - db
      - redis
      - createbuckets
```

● MinIOに初期バケットを作成する

createbucketsコンテナーは、AWS S3の互換サーバーであるMinIO（minio）を操作するためのコンテナーです。初期化処理として、E2Eテストが開始する前に、アプリケーションで使用している画像アップロード先のバケットを作成します。MinIO公式のDocker Imageであるminio/mcを使用すると、CLIからMinIOクライアントを操作するコマンドが使用できます（リストB-11）。

▶ リストB-11　docker-compose.e2e.yaml

```yaml
services:
  db: …
  redis: …
  minio: …
  createbuckets:
    image: minio/mc
    depends_on:
      - minio
    entrypoint: >
      /bin/sh -c "
```

```
      /usr/bin/mc alias set myminio http://minio:9000 root password;
      /usr/bin/mc mb myminio/image --region=ap-northeast-1;
      /usr/bin/mc anonymous set public myminio/image;
      tail -f /dev/null;
      "
  db: …
```

起動時に実行されるentrypointでは、次のMinIOクライアントコマンドを実行しています。

- alias set myminio http://minio:9000 root password;
  - http://minio:9000にmyminioという名称でエイリアスを作成する
- mb myminio/image --region=ap-northeast-1
  - myminioにimageという名称でバケットを作成する
- anonymous set public myminio/image;
  - imageバケットにパブリックアクセス権限を設定する
- tail -f /dev/null;
  - プロセスが終了しないよう待機する

docker compose upのオプションである--exit-code-fromは、いずれかのコンテナーが終了したタイミングでプロセスが終了してしまいます。そのため、バケット初期化完了後もこのコンテナーが終了しないようにtail -f /dev/null;でプロセスを維持します。

docker composeの--exit-code-fromオプションは、特定コンテナーの終了シグナルを返す指定です。npm scriptのdocker:e2e:ciにe2eを指定しているのはこのためです。このオプションは--abort-on-container-exitの指定を暗に含んでしまうため、コンテナーが1つでも停止したら全てのコンテナーが停止してしまい、E2Eテストが実施できません。特定のコンテナーの終了シグナルを無視する指定が現状できないため、ワークアラウンドですがこのようにしています。

●外部コンテナーホストをE2Eコンテナーにマッピングする

GitHub Actionsが実行される仮想マシンでは、ローカル開発環境とは異なりlocalhostや0.0.0.0といったホスト名が利用できません。代わりにコンテナー名を参照することで、コンテナー間通信が可能になります。例えばDATABASE_URLは、localhostの代わりにdbというコンテナー名を使用します（リストB-12）。

▶ リストB-12　docker-compose.e2e.yaml

```yaml
version: "3"
services:
  e2e:
    build:
      context: .
      dockerfile: Dockerfile.e2e
      args:
        DATABASE_URL: postgresql://root:password@db:5432/app-db?schema=public
    environment:
      REDIS_HOST: redis
      REDIS_PORT: 6379
      AWS_S3_ENDPOINT: http://minio:9000
      AWS_ACCESS_KEY_ID: root
      AWS_SECRET_ACCESS_KEY: password
      DATABASE_URL: postgresql://root:password@db:5432/app-db?schema=public
    depends_on:
      - db
      - redis
      - createbuckets
    ports:
      - "3000:3000"
```

本節で紹介した「docker-compose.e2e.yaml」ファイルの全容は次の通りです（リストB-13）。

▶ リストB-13　docker-compose.e2e.yaml

```yaml
version: "3"
services:
  e2e:
    build:
      context: .
      dockerfile: Dockerfile.e2e
      args:
        DATABASE_URL: postgresql://root:password@db:5432/app-db?schema=public
    environment:
      REDIS_HOST: redis
      REDIS_PORT: 6379
      AWS_S3_ENDPOINT: http://minio:9000
      AWS_ACCESS_KEY_ID: root
      AWS_SECRET_ACCESS_KEY: password
      DATABASE_URL: postgresql://root:password@db:5432/app-db?schema=public
    depends_on:
      - db
```

```
      - redis
      - minio
    ports:
      - "3000:3000"
  redis:
    image: redis:latest
    ports:
      - 6379:6379
  minio:
    image: minio/minio:latest
    environment:
      MINIO_ROOT_USER: root
      MINIO_ROOT_PASSWORD: password
    command: server --console-address ":9001" /data
    ports:
      - 9000:9000
      - 9001:9001
  createbuckets:
    image: minio/mc
    depends_on:
      - minio
    entrypoint: >
      /bin/sh -c "
      /usr/bin/mc alias set myminio http://minio:9000 root password;
      /usr/bin/mc mb myminio/image --region=ap-northeast-1;
      /usr/bin/mc anonymous set public myminio/image;
      tail -f /dev/null;
      "
  db:
    image: postgres:13.3
    environment:
      POSTGRES_USER: root
      POSTGRES_PASSWORD: password
    ports:
      - 5432:5432
```