GH-200
GitHub Actionsを使用してワークフローを自動化する

本講義は以下のMicrosoft Learn教材に準拠しています。

https://learn.microsoft.com/ja-jp/training/courses/gh-200t00

COURSE

Automate your workflow with GitHub Actions

Course GH-200T00-A: Automate your workflow with GitHub Actions

本コースについて

- 「GitHub Actionsの利用をこれから開始したい」という方向けのコースです

- このコースではGitHub Actionsの概要、GitHub Actionsを使った継続的インテグレーション（Continuous Integration、CI）と継続的デリバリー（Continuous Delivery、CD）の実装などについて学びます

前提条件

- 以下の知識を持っていると講義がスムーズに理解できます
  - **GitHubの基礎知識**（アカウント、リポジトリ、ブランチ、Issue、プルリクエストなど）
  - ソフトウェア開発の基礎知識（ビルド、テスト、デプロイ、リント、Dockerコンテナー、JavaScript、Python、YAML、パッケージなど）
  - Linuxの基礎知識（Ubuntu、一般的なLinuxコマンド、bashなど）
  - Azureの基礎知識（Azure App Serviceなど）

- **GitHub の基礎知識**については、必要に応じて別コース「GH-900（GitHub の基礎）」をご受講ください

関連コースのご紹介

- GitHub の基礎、GitHub Copilot、GitHub Advanced Security、GitHubの管理については別コースで詳しく解説しています。（本コースではこれらについては解説しません）

- GH-900 GitHub の基礎
- GH-300 GitHub Copilot
- GH-500 GitHub Advanced Security
- GH-100 GitHub Administrator

- 本コースと合わせて、ご受講をご検討ください

目次

- ラーニングパス1
  - モジュール1 GitHub Actions を使用して開発タスクを自動化する
  - モジュール2 GitHub Actions を使用して継続的インテグレーション ワークフローを構築する
  - モジュール3 GitHub Actions を使ったアプリケーションのビルドと Azure へのデプロイ
  - モジュール4 GitHub スクリプトを使用した GitHub の自動化
- ラーニングパス2
  - モジュール5 GitHub Actions を活用して GitHub Packages に公開する
  - モジュール6 カスタム GitHub アクションを作成して公開する
  - モジュール7 企業で GitHub Actions を管理する

目次

モジュール1

- GitHub Actionsとは？
- GitHub Actionsによる自動化の例
- GitHub Actionsの料金
- ワークフローとは？
- イベントとは？
- ジョブとは？
- ステップとは？
- アクションとは？
- アクションを見つけるには？
- ランナーとは？
- まとめ

モジュール1

- GitHub Actionsとは？
- GitHub Actionsによる自動化の例
- GitHub Actionsの料金
- ワークフローとは？
- イベントとは？
- ジョブとは？
- ステップとは？
- アクションとは？
- アクションを見つけるには？
- ランナーとは？
- まとめ

GitHub Actionsとは？

- GitHub上で作業の自動化を行うためのしくみ
- **ワークフロー**（YAMLファイル）で作業を定義する
  - ワークフローはGitHubリポジトリ内に保存される
- ワークフローは**イベント**によって起動される
  - 例えば「リポジトリにコードがプッシュされた」といったイベントにより、ワークフローが起動される
- ワークフロー（正確にはワークフロー内の各ジョブ）は**ランナー**と呼ばれるサーバー上で実行される
  - GitHubホステッドランナー: GitHubが運用するランナー
  - セルフホステッドランナー: ユーザーが運用するランナー

モジュール1

- GitHub Actionsとは？
- GitHub Actionsによる自動化の例
- GitHub Actionsの料金
- ワークフローとは？
- イベントとは？
- ジョブとは？
- ステップとは？
- アクションとは？
- アクションを見つけるには？
- ランナーとは？
- まとめ

GitHub Actionsによる自動化の例

作業の例	説明
コードのビルドとテスト	リポジトリへのコードのプッシュや、プルリクエスト作成時に、自動でビルド・テストを実行
アプリケーションのデプロイ	AWS、 Azure、 Google Cloudなどへ自動でアプリをデプロイ
パッケージの公開	GitHub Packages、 GitHub Container Registry（GHCR）、 npm、PyPI、 Docker Hubなどへパッケージ やDockerイメージを自動でプッシュ
コードの静的解析とLintチェック	ESLint、 Prettier、 flake8などでコードを解析、問題を検出
不要なブランチやキャッシュのクリーンアップ	定期的に古いブランチやビルドキャッシュを削除
ドキュメントの自動生成	コードコメントからドキュメントを生成して更新
SlackやDiscordへの通知の送信	ビルド結果やPR作成をチームに通知
Issueやプルリクエストへの自動ラベル付け・コメント	条件に応じてラベルを付与したりコメントを投稿する

モジュール1

- GitHub Actionsとは？
- GitHub Actionsによる自動化の例
- GitHub Actionsの料金
- ワークフローとは？
- イベントとは？
- ジョブとは？
- ステップとは？
- アクションとは？
- アクションを見つけるには？
- ランナーとは？
- まとめ

GitHub Actionsの料金

- 基本的には有料のサービス
  - 詳細は GitHub Actions の課金 - GitHub ドキュメント を参照

- ただし条件付きで**無料でも使える**
  - セルフホステッドランナーを使用する場合は無料
  - パブリックリポジトリで、標準のGitHubホステッドランナーを使用する場合は無料
  - プライベートリポジトリで、GitHubアカウントが毎月受け取る「クォータ」の範囲で、標準のGitHubホステッドランナーを使用する場合は無料
    - たとえば「GitHub Free」プランでは毎月2,000分の「クォータ」が与えられる

- ※GitHubホステッドランナーとセルフホステッドランナーの詳細は最終モジュールで解説

モジュール1

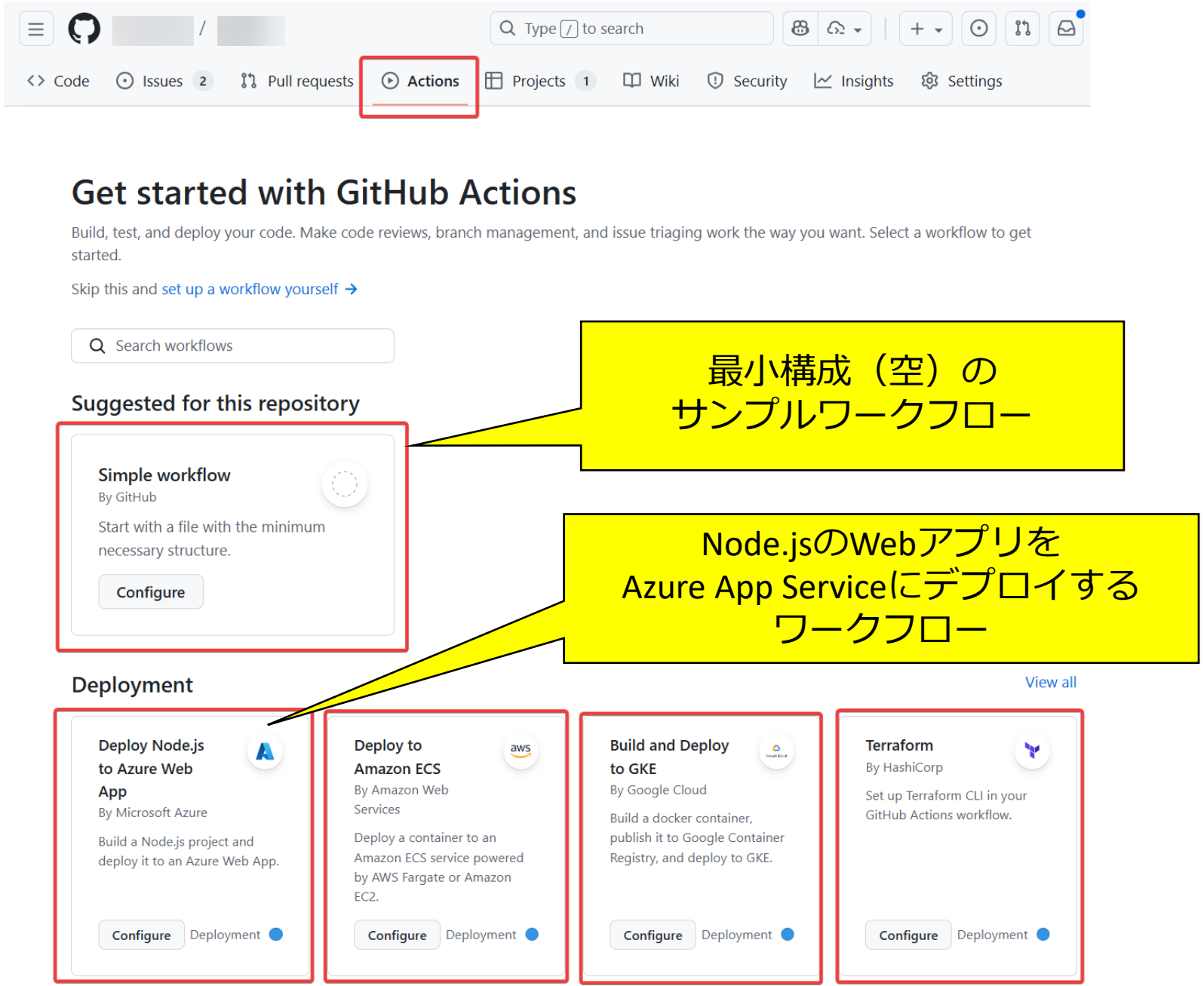
- GitHub Actionsとは？
- GitHub Actionsによる自動化の例
- GitHub Actionsの料金
- ワークフローとは？
- イベントとは？
- ジョブとは？
- ステップとは？
- アクションとは？
- アクションを見つけるには？
- ランナーとは？
- まとめ

ワークフローとは？

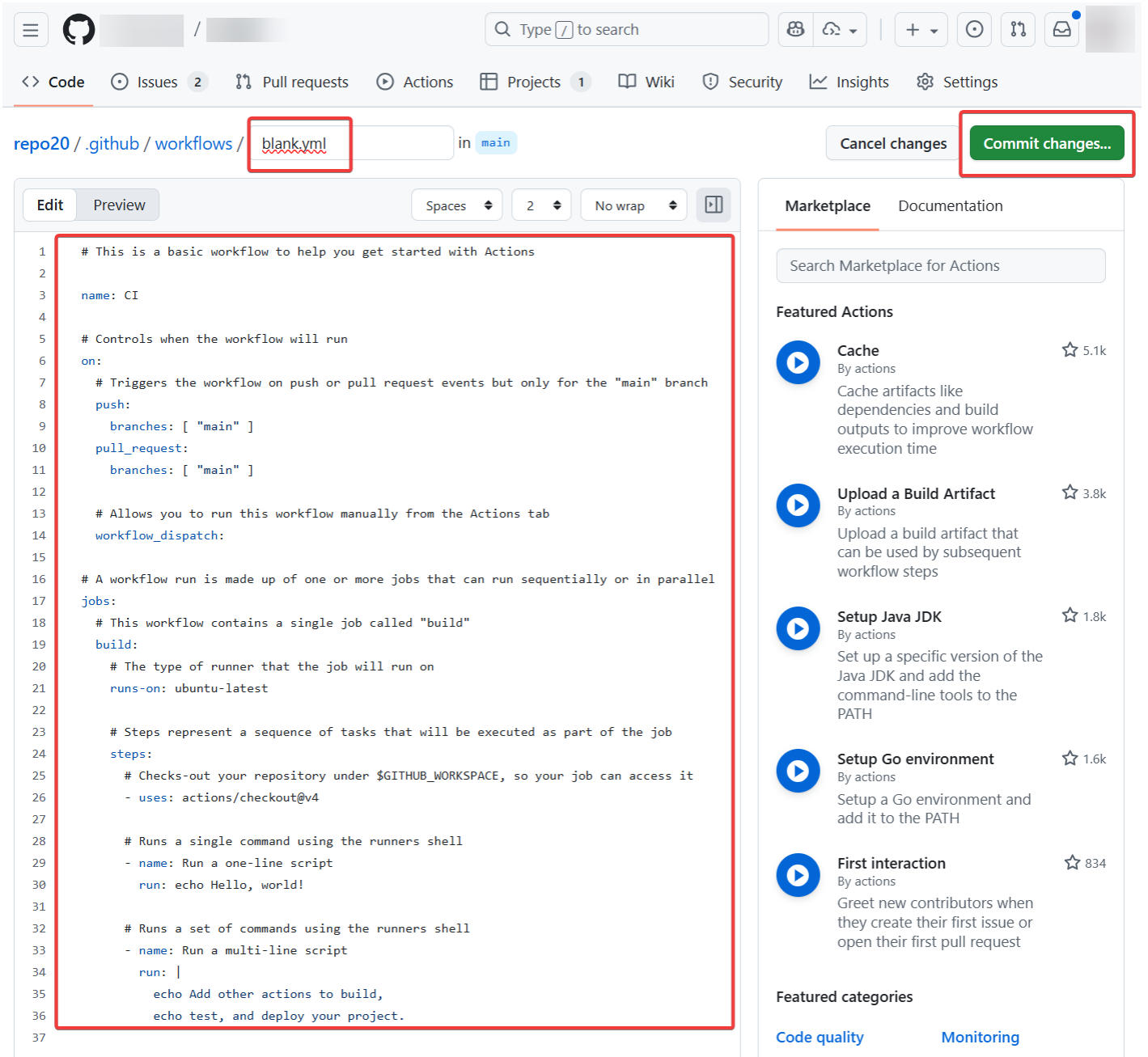
- GitHub Actionsの自動化の手続きを定義するファイル
- YAML形式
- GitHubリポジトリの `.github/workflows` ディレクトリに保存される
- ファイル名は任意（～.yml）
- 一つのGitHubリポジトリに複数のワークフローを配置できる
- `#` で始まる行はコメント

Workflows - GitHub ドキュメント

リポジトリの「Actions」をクリックすると、多数のテンプレートが表示される。
使用したいテンプレートの「Configure」ボタンをクリック。



ワークフローのYAMLファイルが準備される。
YAMLファイルの内容やファイル名を変更し、「Commit changes...」で保存。



モジュール1

- GitHub Actionsとは？
- GitHub Actionsによる自動化の例
- GitHub Actionsの料金
- ワークフローとは？
- イベントとは？
- ジョブとは？
- ステップとは？
- アクションとは？
- アクションを見つけるには？
- ランナーとは？
- まとめ

イベントとは？

- ワークフロー実行をトリガーする、リポジトリ内の特定のアクティビティ
- 例えば「リポジトリのブランチにコードがプッシュされた」、「Issueにコメントが追加された」「プルリクエストが作成された」といったアクティビティ
- イベントにより、ワークフローが起動される

- どのイベントでワークフローをトリガーするかは、ワークフローの「`on:`」で定義

イベントの例（1）

```yaml
name: Add Kaomoji to Issue Comments

on:
  issue_comment:
    types: [created]

permissions:
  issues: write

jobs:
  add-kaomoji:
    runs-on: ubuntu-latest
    steps:
      - name: Add Kaomoji
        uses: actions/github-script@v7
        with:
          script: |
            const kaomojis = ["( ^ ▽ ^ )", "(ノ ✧▽✧)ノ ", "٩( ´ ᗜ ` *)و", "(•••)و"];
            const random = kaomojis[Math.floor(Math.random() * kaomojis.length)];
            github.rest.issues.createComment({
              issue_number: context.issue.number,
              owner: context.repo.owner,
              repo: context.repo.repo,
              body: `${random}`
            });
```

このワークフローを動かす「イベント」の指定。
この例では「**Issueのコメントが作成されたとき**」と指定している。

イベントの例（2）

```yaml
name: Python application

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

permissions:
  contents: read

jobs:
  build:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v4
    - name: Set up Python 3.10
      uses: actions/setup-python@v3
      with:
        python-version: "3.10"
    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install flake8 pytest
        if [ -f requirements.txt ]; then pip install -r requirements.txt; fi
    - name: Lint with flake8
      run: |
        # stop the build if there are Python syntax errors or undefined names
        flake8 . --count --select=E9,F63,F7,F82 --show-source --statistics
        # exit-zero treats all errors as warnings. The GitHub editor is 127 chars wide
        flake8 . --count --exit-zero --max-complexity=10 --max-line-length=127 --statistics
    - name: Test with pytest
      run: |
        pytest
```

リポジトリの main ブランチに
プッシュされた際、
または、
プルリクエストが作成された際に、
このワークフローを実行する。

このように複数のイベントを指定することもできる

モジュール1

- GitHub Actionsとは？
- GitHub Actionsによる自動化の例
- GitHub Actionsの料金
- ワークフローとは？
- イベントとは？
- ジョブとは？
- ステップとは？
- アクションとは？
- アクションを見つけるには？
- ランナーとは？
- まとめ

ジョブとは？

- 1つ～複数のステップの集まり
- 各ジョブは別のランナー上で実行される
  - ジョブ1はランナーAで、ジョブ2はランナーBで・・・
- 複数のジョブがある場合、デフォルトでは、並列で実行されていく
  - たとえば、コードをWindowsとLinuxの各環境でテストする場合、ジョブを使用することで、複数のテストを並列実行できる

  - **並列で実行されるジョブのうち一つが失敗しても、別のジョブはその影響を受けず、実行される**

ワークフローでジョブを使用する - GitHub ドキュメント

ここでは test というIDのジョブを定義している

```yaml
name: Matrix Example

on: [push]

jobs:
  test:
    runs-on: ${{ matrix.os }}   # ← OSを切り替え
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest, macos-latest]   # 複数プラットフォーム
        node: [16, 18, 20]                                  # 複数バージョン
    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: ${{ matrix.node }}

      - name: Install dependencies
        run: npm install

      - name: Run tests
        run: npm test
```

```yaml
name: Matrix Example

on: [push]

jobs:
  test:
    runs-on: ${{ matrix.os }}   # ← OSを切り替え
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest, macos-latest]   # 複数プラットフォーム
        node: [16, 18, 20]                                  # 複数バージョン
    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: ${{ matrix.node }}

      - name: Install dependencies
        run: npm install

      - name: Run tests
        run: npm test
```

ただし、ここで strategy: matrix: を使用して、3種類のOS、3種類のNode.jsバージョンの組み合わせを作っている。
つまり 3 x 3 = 9 ジョブが実際には作られる

この9ジョブはそれぞれ別ランナー上で実行され、あるジョブの失敗は別のジョブの実行に影響を与えない。

```yaml
name: Matrix Example

on: [push]

jobs:
  test:
    runs-on: ${{ matrix.os }}   # ← OSを切り替え
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest, macos-latest]   # 複数プラットフォーム
        node: [16, 18, 20]                                  # 複数バージョン
    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: ${{ matrix.node }}

      - name: Install dependencies
        run: npm install

      - name: Run tests
        run: npm test
```

matrixのOSを参照

matrixのNode.jsバージョンを参照

ジョブの依存関係

- オプションでジョブに「依存関係」を指定することもできる
  - `needs:`を使用。

- **依存関係が指定されている場合は、前のジョブが失敗すると後続ジョブは実行されない**
  - この例では、ジョブ「build」（ビルドとテスト）が正常に完了した場合のみ、ジョブ「deploy」が実行される

```yaml
name: job example
on: [push]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: build
        run: ...
      - name: test
        run: ...
  deploy:
    runs-on: ubuntu-latest
    needs: build
    steps:
      - run: ...
```

ワークフローでジョブを使用する - GitHub ドキュメント

ワークフローの例

```yaml
name: Add Kaomoji to Issue Comments

on:
  issue_comment:
    types: [created]

permissions:
  issues: write

jobs:
  add-kaomoji:
    runs-on: ubuntu-latest
    steps:
      - name: Add Kaomoji
        uses: actions/github-script@v7
        with:
          script: |
            const kaomojis = ["( ^ ▽ ^ )", "(ノ ✧▽✧)ノ ", "٩( ´ ᗝ ` *)و", "(•••)و"];
            const random = kaomojis[Math.floor(Math.random() * kaomojis.length)];
            github.rest.issues.createComment({
              issue_number: context.issue.number,
              owner: context.repo.owner,
              repo: context.repo.repo,
              body: `${random}`
            });
```

このワークフローで実行する
「ジョブ」を指定

ワークフローの例

```yaml
name: Add Kaomoji to Issue Comments

on:
  issue_comment:
    types: [created]

permissions:
  issues: write

jobs:
  add-kaomoji:
    runs-on: ubuntu-latest
    steps:
      - name: Add Kaomoji
        uses: actions/github-script@v7
        with:
          script: |
            const kaomojis = ["( ^ ▽ ^ )", "(ノ ✧▽✧)ノ ", "٩( ´ ᗜ ` *)و", "(•••)و"];
            const random = kaomojis[Math.floor(Math.random() * kaomojis.length)];
            github.rest.issues.createComment({
              issue_number: context.issue.number,
              owner: context.repo.owner,
              repo: context.repo.repo,
              body: `${random}`
            });
```

各ジョブは一意の識別子「ジョブID」を持つ
（オプションで、name: を使用して、UIに表示されるわかりやすい名前も指定できる）

ワークフローの例

```yaml
name: Add Kaomoji to Issue Comments

on:
  issue_comment:
    types: [created]

permissions:
  issues: write

jobs:
  add-kaomoji:
    runs-on: ubuntu-latest
    steps:
      - name: Add Kaomoji
        uses: actions/github-script@v7
        with:
          script: |
            const kaomojis = ["( ^ ▽ ^ )", "(ノ ✧▽✧)ノ ", "٩( ´ ᗜ ` *)و", "(•••)و"];
            const random = kaomojis[Math.floor(Math.random() * kaomojis.length)];
            github.rest.issues.createComment({
              issue_number: context.issue.number,
              owner: context.repo.owner,
              repo: context.repo.repo,
              body: `${random}`
            });
```

各ジョブはランナー上で実行される。
ここではGitHub ホステッドランナーの ubuntu-latest を指定している

モジュール1

- GitHub Actionsとは？
- GitHub Actionsによる自動化の例
- GitHub Actionsの料金
- ワークフローとは？
- イベントとは？
- ジョブとは？
- ステップとは？
- アクションとは？
- アクションを見つけるには？
- ランナーとは？
- まとめ

ステップとは？

- ジョブ内の手続きの指定
- 1つのジョブに1つ～複数のステップを指定できる
- 各ステップは上から順に、連続的に実行されていく
- 各ステップではコマンド、シェルスクリプト、アクションなどを実行できる

- **あるステップが失敗した場合は、その続きのステップは実行されない**

ただし、例外として以下の指定がある場合は続行できます：

- `continue-on-error: true` を付けると、そのステップが失敗してもジョブは継続します。
- `if: ${{ always() }}` を条件にすると、失敗に関係なくそのステップは必ず実行されます。

Workflows - GitHub ドキュメント

ワークフローの例

```yaml
name: Python application

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

permissions:
  contents: read

jobs:
  build:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v4
    - name: Set up Python 3.10
      uses: actions/setup-python@v3
      with:
        python-version: "3.10"
    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install flake8 pytest
        if [ -f requirements.txt ]; then pip install -r requirements.txt; fi
    - name: Lint with flake8
      run: |
        # stop the build if there are Python syntax errors or undefined names
        flake8 . --count --select=E9,F63,F7,F82 --show-source --statistics
        # exit-zero treats all errors as warnings. The GitHub editor is 127 chars wide
        flake8 . --count --exit-zero --max-complexity=10 --max-line-length=127 --statistics
    - name: Test with pytest
      run: |
        pytest
```

このワークフローにGitHubリポジトリの読み取り権限を与える

ワークフローの例

```yaml
name: Python application

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

permissions:
  contents: read

jobs:
  build:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v4
    - name: Set up Python 3.10
      uses: actions/setup-python@v3
      with:
        python-version: "3.10"
    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install flake8 pytest
        if [ -f requirements.txt ]; then pip install -r requirements.txt; fi
    - name: Lint with flake8
      run: |
        # stop the build if there are Python syntax errors or undefined names
        flake8 . --count --select=E9,F63,F7,F82 --show-source --statistics
        # exit-zero treats all errors as warnings. The GitHub editor is 127 chars wide
        flake8 . --count --exit-zero --max-complexity=10 --max-line-length=127 --statistics
    - name: Test with pytest
      run: |
        pytest
```

ジョブの定義。
ここでジョブIDは「build」としている

ワークフローの例

```yaml
name: Python application

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

permissions:
  contents: read

jobs:
  build:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v4
    - name: Set up Python 3.10
      uses: actions/setup-python@v3
      with:
        python-version: "3.10"
    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install flake8 pytest
        if [ -f requirements.txt ]; then pip install -r requirements.txt; fi
    - name: Lint with flake8
      run: |
        # stop the build if there are Python syntax errors or undefined names
        flake8 . --count --select=E9,F63,F7,F82 --show-source --statistics
        # exit-zero treats all errors as warnings. The GitHub editor is 127 chars wide
        flake8 . --count --exit-zero --max-complexity=10 --max-line-length=127 --statistics
    - name: Test with pytest
      run: |
        pytest
```

このジョブは
ubuntu-latest
というGitHubホステッドランナー上で
実行される

ワークフローの例

```yaml
name: Python application

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

permissions:
  contents: read

jobs:
  build:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v4
    - name: Set up Python 3.10
      uses: actions/setup-python@v3
      with:
        python-version: "3.10"
    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install flake8 pytest
        if [ -f requirements.txt ]; then pip install -r requirements.txt; fi
    - name: Lint with flake8
      run: |
        # stop the build if there are Python syntax errors or undefined names
        flake8 . --count --select=E9,F63,F7,F82 --show-source --statistics
        # exit-zero treats all errors as warnings. The GitHub editor is 127 chars wide
        flake8 . --count --exit-zero --max-complexity=10 --max-line-length=127 --statistics
    - name: Test with pytest
      run: |
        pytest
```

このジョブのステップの定義

ワークフローの例

このジョブには5つのステップがあり、
一つのランナー上で、上から順に実行されていく

```
  steps:
  - uses: actions/checkout@v4
  - name: Set up Python 3.10
    uses: actions/setup-python@v3
    with:
      python-version: "3.10"
  - name: Install dependencies
    run: |
      python -m pip install --upgrade pip
      pip install flake8 pytest
      if [ -f requirements.txt ]; then pip install -r requirements.txt; fi
  - name: Lint with flake8
    run: |
      # stop the build if there are Python syntax errors or undefined names
      flake8 . --count --select=E9,F63,F7,F82 --show-source --statistics
      # exit-zero treats all errors as warnings. The GitHub editor is 127 chars wide
      flake8 . --count --exit-zero --max-complexity=10 --max-line-length=127 --statistics
  - name: Test with pytest
    run: |
      pytest
```

ワークフローの例

```
steps:
- uses: actions/checkout@v4
- name: Set up Python 3.10
  uses: actions/setup-python@v3
  with:
    python-version: "3.10"
- name: Install dependencies
  run: |
    python -m pip install --upgrade pip
    pip install flake8 pytest
    if [ -f requirements.txt ]; then pip install -r requirements.txt; fi
- name: Lint with flake8
  run: |
    # stop the build if there are Python syntax errors or undefined names
    flake8 . --count --select=E9,F63,F7,F82 --show-source --statistics
    # exit-zero treats all errors as warnings. The GitHub editor is 127 chars wide
    flake8 . --count --exit-zero --max-complexity=10 --max-line-length=127 --statistics
- name: Test with pytest
  run: |
    pytest
```

uses: はアクションの指定。
actions/checkout@4アクションで、
このワークフローのGitHubリポジトリの
ソースコードをランナーにコピーする

ワークフローの例

setup-pythonアクションで、
ランナーにPython 3.10 をインストールする。
with: はアクションに対するパラメータの指定。

```
steps:
- uses: actions/checkout@v4
- name: Set up Python 3.10
  uses: actions/setup-python@v3
  with:
    python-version: "3.10"
- name: Install dependencies
  run: |
    python -m pip install --upgrade pip
    pip install flake8 pytest
    if [ -f requirements.txt ]; then pip install -r requirements.txt; fi
- name: Lint with flake8
  run: |
    # stop the build if there are Python syntax errors or undefined names
    flake8 . --count --select=E9,F63,F7,F82 --show-source --statistics
    # exit-zero treats all errors as warnings. The GitHub editor is 127 chars wide
    flake8 . --count --exit-zero --max-complexity=10 --max-line-length=127 --statistics
- name: Test with pytest
  run: |
    pytest
```

ワークフローの例

```
steps:
- uses: actions/checkout@v4
- name: Set up Python 3.10
  uses: actions/setup-python@v3
  with:
    python-version: "3.10"
- name: Install dependencies
  run: |
    python -m pip install --upgrade pip
    pip install flake8 pytest
    if [ -f requirements.txt ]; then pip install -r requirements.txt; fi
- name: Lint with flake8
  run: |
    # stop the build if there are Python syntax errors or undefined names
    flake8 . --count --select=E9,F63,F7,F82 --show-source --statistics
    # exit-zero treats all errors as warnings. The GitHub editor is 127 chars wide
    flake8 . --count --exit-zero --max-complexity=10 --max-line-length=127 --statistics
- name: Test with pytest
  run: |
    pytest
```

pipコマンドをインストール、
pipで必要なPythonパッケージをインストール。

run: は、指定したコマンドを
（この場合はubuntuランナー上で）実行する

ワークフローの例

```
    steps:
    - uses: actions/checkout@v4
    - name: Set up Python 3.10
      uses: actions/setup-python@v3
      with:
        python-version: "3.10"
    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install flake8 pytest
        if [ -f requirements.txt ]; then pip install -r requirements.txt; fi
    - name: Lint with flake8
      run: |
        # stop the build if there are Python syntax errors or undefined names
        flake8 . --count --select=E9,F63,F7,F82 --show-source --statistics
        # exit-zero treats all errors as warnings. The GitHub editor is 127 chars wide
        flake8 . --count --exit-zero --max-complexity=10 --max-line-length=127 --statistics
    - name: Test with pytest
      run: |
        pytest
```

flake8 （リンター）を使用して、Python コードが一般的なコーディング規約に沿って記述されていることをチェック

ワークフローの例

```
steps:
- uses: actions/checkout@v4
- name: Set up Python 3.10
  uses: actions/setup-python@v3
  with:
    python-version: "3.10"
- name: Install dependencies
  run: |
    python -m pip install --upgrade pip
    pip install flake8 pytest
    if [ -f requirements.txt ]; then pip inst          quirements.txt; fi
- name: Lint with flake8
  run: |
    # stop the build if there are Pytho      ax errors or undefined names
    flake8 . --count --select=E9,F63      82 --show-source --statistics
    # exit-zero treats all errors      warnings. The GitHub editor is 127 chars wide
    flake8 . --count --exit-zero      -max-complexity=10 --max-line-length=127 --statistics
- name: Test with pytest
  run: |
    pytest
```

pytest（テストフレームワーク）を使用して、
テストを実行

モジュール1

- GitHub Actionsとは？
- GitHub Actionsによる自動化の例
- GitHub Actionsの料金
- ワークフローとは？
- イベントとは？
- ジョブとは？
- ステップとは？
- アクションとは？
- アクションを見つけるには？
- ランナーとは？
- まとめ

アクションとは？

- 「GitHub Actions」と「アクション」は**別のもの**を指すので注意！
- 「GitHub Actions」はGitHubが提供する**自動化のサービス**
- 「アクション」は「GitHub Actions」の中の**定義済みの処理**
- 「アクション」は「aaa/bbb@v2」といった名前とバージョンで表される

GitHub Actions を理解する - GitHub ドキュメント

アクションとは？

- アクションは、ワークフローの中で実行される、あらかじめ定義された作業
- 例:
  - GitHubリポジトリからランナーへコードをコピーする（チェックアウト） `actions/checkout@4`
  - ランタイムやコマンドをランナーにインストールする `actions/setup-python@v6`
  - ビルドされた成果物をサーバーやAzureにデプロイする `azure/weapps-deploy@v2`
  - JavaScriptを実行する `actions/github-script@v8`
    - JavaScriptを使って、GitHubに対する操作を実行できる

GitHub Actions を理解する - GitHub ドキュメント

アクションの例: `actions/checkout@4`

- GitHubリポジトリからランナーへとコードを取り出す「チェックアウト」を実行
- **ほとんどのワークフローの最初でこれを使用する**
- 内部的には git init, git remote add, git fetch などのgitコマンドを組み合わせてチェックアウトを行う
- アクションは「`actions/checkout@4`」といった形の名前を持つ。
  - `actions` … GitHubのorganization名
  - `checkout` … actions以下のリポジトリの名前
  - @v4 … メジャーバージョンタグ

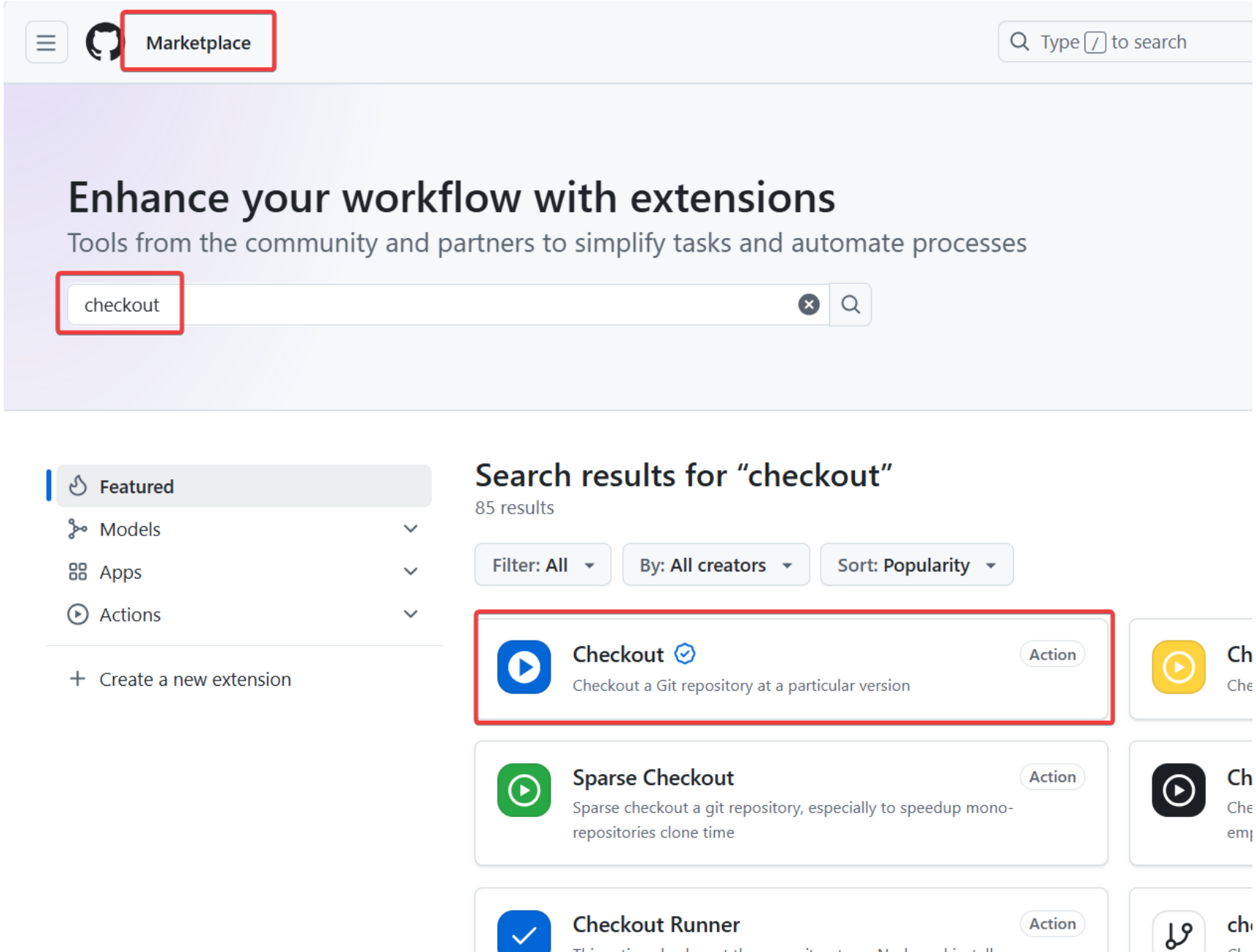
モジュール1

- GitHub Actionsとは？
- GitHub Actionsによる自動化の例
- GitHub Actionsの料金
- ワークフローとは？
- イベントとは？
- ジョブとは？
- ステップとは？
- アクションとは？
- アクションを見つけるには？
- ランナーとは？
- まとめ

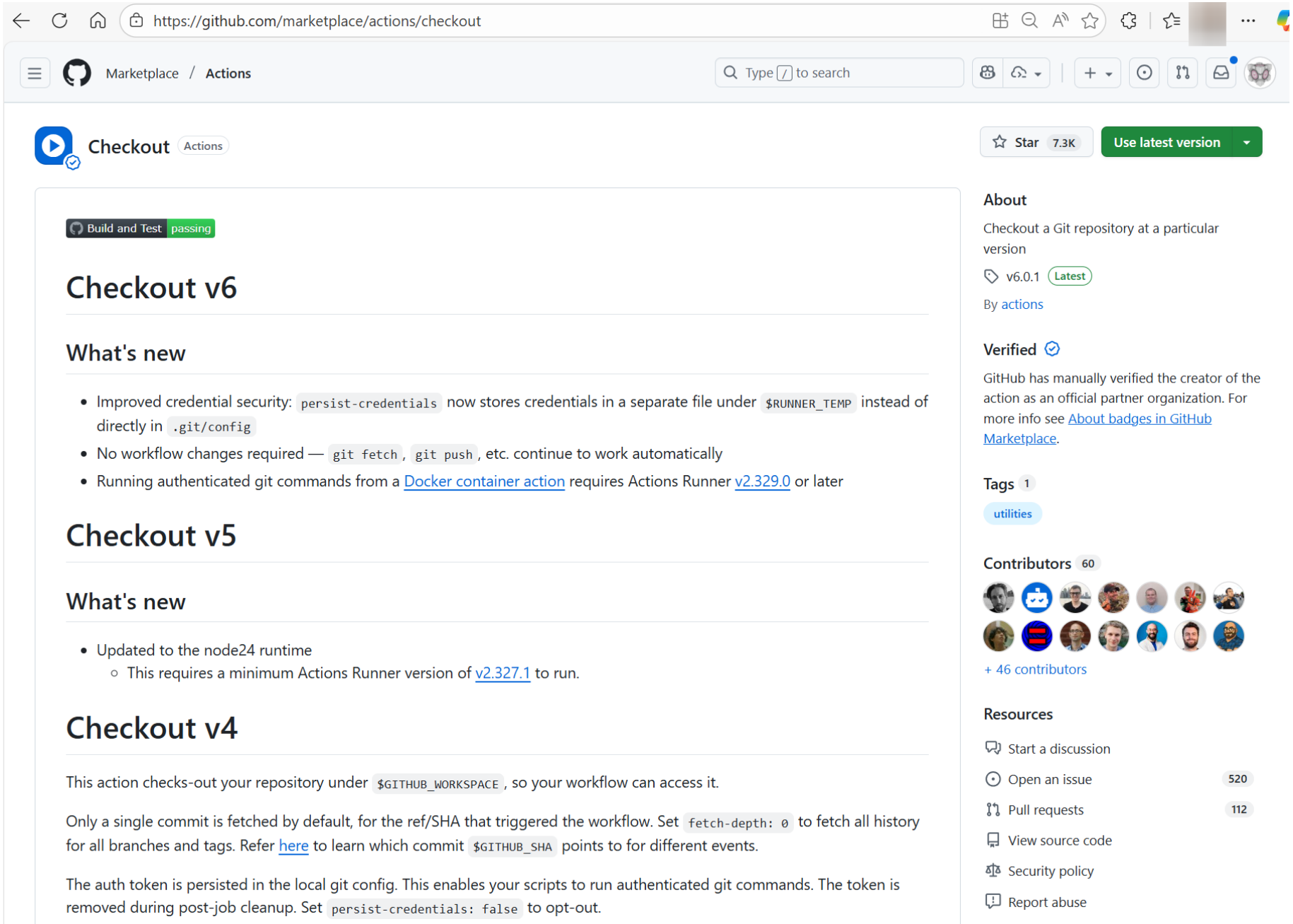
アクションを見つけるには？

- 「GitHub Marketplace」を使用する
- GitHubで使用できるAIモデル、アプリ、GitHub Actionsの「アクション」などを探せる場所
- https://github.com/marketplace

GitHub Marketplaceでアクションを探す例



actions/checkoutのリポジトリ（@v4, @v5, @v6といったメジャーバージョンがあることがわかる）



actions/checkout@4 の動作例


```
1  ▶ Run actions/checkout@v4
16 Syncing repository: hiryamada/repo20
17 ▶ Getting Git version info
21 Temporarily overriding HOME='/home/runner/work/_temp/74f3d5f1-004c-4406-aef2-7243809f53e9' before making global git config changes
22 Adding repository directory to the temporary git global config as a safe directory
23 /usr/bin/git config --global --add safe.directory /home/runner/work/repo20/repo20
24 Deleting the contents of '/home/runner/work/repo20/repo20'
25 ▼ Initializing the repository
26   /usr/bin/git init /home/runner/work/repo20/repo20
27   hint: Using 'master' as the name for the initial branch. This default branch name
28   hint: will change to "main" in Git 3.0. To configure the initial branch name
29   hint: to use in all of your new repositories, which will suppress this warning,
30   hint: call:
31   hint:
32   hint:         git config --global init.defaultBranch <name>
33   hint:
34   hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
35   hint: 'development'. The just-created branch can be renamed via this command:
36   hint:
37   hint:         git branch -m <name>
38   hint:
39   hint: Disable this message with "git config set advice.defaultBranchName false"
40   Initialized empty Git repository in /home/runner/work/repo20/repo20/.git/
41   /usr/bin/git remote add origin https://github.com/hiryamada/repo20
42 ▶ Disabling automatic garbage collection
44 ▶ Setting up auth
52 ▼ Fetching the repository
53   /usr/bin/git -c protocol.version=2 fetch --no-tags --prune --no-recurse-submodules --depth=1 origin
     +b8948e3ccc8211de714cb0f2f1b185f629042afa:refs/remotes/origin/main
54   From https://github.com/hiryamada/repo20
55    * [new ref]         b8948e3ccc8211de714cb0f2f1b185f629042afa -> origin/main
56 ▶ Determining the checkout info
57 /usr/bin/git sparse-checkout disable
58 /usr/bin/git config --local --unset-all extensions.worktreeConfig
59 ▶ Checking out the ref
63 /usr/bin/git log -1 --format=%H
64 b8948e3ccc8211de714cb0f2f1b185f629042afa
```

モジュール1

- GitHub Actionsとは？
- GitHub Actionsによる自動化の例
- GitHub Actionsの料金
- ワークフローとは？
- イベントとは？
- ジョブとは？
- ステップとは？
- アクションとは？
- アクションを見つけるには？
- ランナーとは？
- まとめ

ランナーとは？

- ジョブを実行するためのサーバー
- ワークフロー内の各ジョブはそれぞれ別のランナー上で実行される

- ランナーの種類
  - GitHubホステッドランナー: GitHubが運用するランナー
  - セルフホステッドランナー: ユーザーが運用するランナー

セルフホステッド ランナー - GitHub ドキュメント

GitHub ホステッド ランナー - GitHub ドキュメント

GitHubホステッドランナー

- GitHubホステッドランナーの実態はAzure 仮想マシン（VM）である。

- https://github.com/actions/runner-images リポジトリに、利用可能なランナー（正確にはVMのイメージ）の一覧がある

GitHub ホステッド ランナー - GitHub ドキュメント

利用可能なランナー（VMイメージ）

ubuntu-latest
は一般的なジョブの実行によく使用される

Available Images

Image	YAML Label	Included Software
Ubuntu 24.04	`ubuntu-latest` or `ubuntu-24.04`	ubuntu-24.04
Ubuntu 22.04	`ubuntu-22.04`	ubuntu-22.04
macOS 26 Arm64 `beta`	`macos-26` or `macos-26-xlarge`	macOS-26-arm64
macOS 15	`macos-latest-large`, `macos-15-large`, or `macos-15-intel`	macOS-15
macOS 15 Arm64	`macos-latest`, `macos-15`, or `macos-15-xlarge`	macOS-15-arm64
macOS 14	`macos-14-large`	macOS-14
macOS 14 Arm64	`macos-14` or `macos-14-xlarge`	macOS-14-arm64
macOS 13 Deprecated	`macos-13` or `macos-13-large`	macOS-13
macOS 13 Arm64 Deprecated	`macos-13-xlarge`	macOS-13-arm64
Windows Server 2025	`windows-latest` or `windows-2025`	windows-2025
Windows Server 2022	`windows-2022`	windows-2022
Windows Server 2019 Deprecated	`windows-2019`	windows-2019

利用可能なランナー（VMイメージ）

ここをクリックすると、このランナー（VMイメージ）にプリインストールされているソフトウェアを確認できる。

Available Images

Image	YAML Label	Included Software
Ubuntu 24.04	`ubuntu-latest` or `ubuntu-24.04`	ubuntu-24.04
Ubuntu 22.04	`ubuntu-22.04`	ubuntu-22.04
macOS 26 Arm64 `beta`	`macos-26` or `macos-26-xlarge`	macOS-26-arm64
macOS 15	`macos-latest-large`, `macos-15-large`, or `macos-15-intel`	macOS-15
macOS 15 Arm64	`macos-latest`, `macos-15`, or `macos-15-xlarge`	macOS-15-arm64
macOS 14	`macos-14-large`	macOS-14
macOS 14 Arm64	`macos-14` or `macos-14-xlarge`	macOS-14-arm64
macOS 13 Deprecated	`macos-13` or `macos-13-large`	macOS-13
macOS 13 Arm64 Deprecated	`macos-13-xlarge`	macOS-13-arm64
Windows Server 2025	`windows-latest` or `windows-2025`	windows-2025
Windows Server 2022	`windows-2022`	windows-2022
Windows Server 2019 Deprecated	`windows-2019`	windows-2019

モジュール1

- GitHub Actionsとは？
- GitHub Actionsによる自動化の例
- GitHub Actionsの料金
- ワークフローとは？
- イベントとは？
- ジョブとは？
- ステップとは？
- アクションとは？
- アクションを見つけるには？
- ランナーとは？
- まとめ

まとめ

- **GitHub Actions**は、GitHub上で作業の自動化を行うしくみ。
- 基本的に有料のサービスだが条件付きで無料でも利用できる。
- **ワークフロー**（YAMLファイル）で作業を定義する。
- ワークフローは例えば「リポジトリにコードがプッシュされた」といった**イベント**によって起動される。
- ワークフロー内には1つ～複数の**ジョブ**が定義される。
  - ジョブは基本的には並列で実行されるが、**依存関係**を設定すれば、ジョブを順番に実行することも可能。
- ジョブの中には1つ～複数の**ステップ**が定義される。
  - ステップを使用して、コードのチェックアウト（リポジトリからランナーへコードを取り出す）、ビルド、テスト、デプロイなどの処理が実行される。
- ワークフロー内の各ジョブは**ランナー**と呼ばれるサーバー上で実行される。

目次

- ラーニングパス1
  - モジュール1 GitHub Actions を使用して開発タスクを自動化する
  - モジュール2 GitHub Actions を使用して継続的インテグレーション ワークフローを構築する
  - モジュール3 GitHub Actions を使ったアプリケーションのビルドと Azure へのデプロイ
  - モジュール4 GitHub スクリプトを使用した GitHub の自動化
- ラーニングパス2
  - モジュール5 GitHub Actions を活用して GitHub Packages に公開する
  - モジュール6 カスタム GitHub アクションを作成して公開する
  - モジュール7 企業で GitHub Actions を管理する

モジュール2

- 継続的インテグレーションとは？
- GitHub Actionsによる継続的インテグレーションの例
- まとめ

モジュール2

- 継続的インテグレーションとは？
- GitHub Actionsによる継続的インテグレーションの例
- まとめ

継続的インテグレーションとは？

- Continuous Integration, CI
- 開発者がコードを頻繁に共有リポジトリへ統合するプロセス
- 自動ビルドや自動テストを実行
- 文法やスタイルのチェック、セキュリティのチェックなどを行うこともできる
- コードの不整合や不具合を早期に発見できる
- 品質を保ち、開発スピードを上げる
- GitHubリポジトリとGitHub Actionsを使用して、CIを実現できる

継続的インテグレーション - GitHub ドキュメント

モジュール2

- 継続的インテグレーションとは？
- GitHub Actionsによる継続的インテグレーションの例
- まとめ

GitHub Actionsによる継続的インテグレーション（CI）の例

```yaml
name: Matrix Example

on: [push]

jobs:
  test:
    runs-on: ${{ matrix.os }}   # ← OSを切り替え
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest, macos-latest]   # 複数プラットフォーム
        node: [16, 18, 20]                                  # 複数バージョン
    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: ${{ matrix.node }}

      - name: Install dependencies
        run: npm install

      - name: Run tests
        run: npm test
```

GitHub Actionsによる継続的インテグレーション（CI）の例

```yaml
name: Matrix Example

on: [push]

jobs:
  test:
    runs-on: ${{ matrix.os }}   # ← OSを切り替え
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest, macos-latest]   # 複数プラットフォーム
        node: [16, 18, 20]                                  # 複数バージョン
    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: ${{ matrix.node }}

      - name: Install dependencies
        run: npm install

      - name: Run tests
        run: npm test
```

このワークフローにわかりやすい名前を付ける
（GitHubのWeb UI上に表示される）

GitHub Actionsによる継続的インテグレーション（CI）の例

```yaml
name: Matrix Example

on: [push]

jobs:
  test:
    runs-on: ${{ matrix.os }}   # ← OSを切り替え
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest, macos-latest]   # 複数プラットフォーム
        node: [16, 18, 20]                                  # 複数バージョン
    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: ${{ matrix.node }}

      - name: Install dependencies
        run: npm install

      - name: Run tests
        run: npm test
```

リポジトリにコードがプッシュされるたびに、このワークフローが実行される

GitHub Actionsによる継続的インテグレーション（CI）の例

ジョブの定義

```yaml
name: Matrix Example

on: [push]

jobs:
  test:
    runs-on: ${{ matrix.os }}   # ← OSを切り替え
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest, macos-latest]   # 複数プラットフォーム
        node: [16, 18, 20]                                  # 複数バージョン
    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: ${{ matrix.node }}

      - name: Install dependencies
        run: npm install

      - name: Run tests
        run: npm test
```

GitHub Actionsによる継続的インテグレーション（CI）の例

```yaml
name: Matrix Example

on: [push]

jobs:
  test:
    runs-on: ${{ matrix.os }}   # ← OSを切り替え
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest, macos-latest]   # 複数プラットフォーム
        node: [16, 18, 20]                                  # 複数バージョン
    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: ${{ matrix.node }}

      - name: Install dependencies
        run: npm install

      - name: Run tests
        run: npm test
```

testというIDのジョブを定義している

GitHub Actionsによる継続的インテグレーション（CI）の例

```yaml
name: Matrix Example

on: [push]

jobs:
  test:
    runs-on: ${{ matrix.os }}   # ← OSを切り替え
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest, macos-latest]   # 複数プラットフォーム
        node: [16, 18, 20]                                  # 複数バージョン
    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: ${{ matrix.node }}

      - name: Install dependencies
        run: npm install

      - name: Run tests
        run: npm test
```

matrixストラテジーを使用して、OS3種類、Node.jsのバージョン3種類の組み合わせ、合計9個のジョブを作り、並列実行する。各ジョブは別のランナー上で実行されていく

GitHub Actionsによる継続的インテグレーション（CI）の例

```yaml
name: Matrix Example

on: [push]

jobs:
  test:
    runs-on: ${{ matrix.os }}   # ← OSを切り替え
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest, macos-latest]   # 複数プラットフォーム
        node: [16, 18, 20]                                  # 複数バージョン
    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: ${{ matrix.node }}

      - name: Install dependencies
        run: npm install

      - name: Run tests
        run: npm test
```

各ジョブでどのOSを使用するかの指定。
ubuntu-latest, windows-latest, macos-latest のいずれかとなる

GitHub Actionsによる継続的インテグレーション（CI）の例

```yaml
name: Matrix Example

on: [push]

jobs:
  test:
    runs-on: ${{ matrix.os }}   # ← OSを切り替え
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest, macos-latest]   # 複数プラットフォーム
        node: [16, 18, 20]                                  # 複数バージョン
    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: ${{ matrix.node }}

      - name: Install dependencies
        run: npm install

      - name: Run tests
        run: npm test
```

ジョブ内のステップの定義

GitHub Actionsによる継続的インテグレーション（CI）の例

```yaml
name: Matrix Example

on: [push]

jobs:
  test:
    runs-on: ${{ matrix.os }}   # ← OSを切り替え
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest, macos-latest]   # 複数プラットフォーム
        node: [16, 18, 20]                                  # 複数バージョン
    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: ${{ matrix.node }}

      - name: Install dependencies
        run: npm install

      - name: Run tests
        run: npm test
```

まずGitHubリポジトリからランナーへとコードを取り出す

GitHub Actionsによる継続的インテグレーション（CI）の例

```yaml
name: Matrix Example

on: [push]

jobs:
  test:
    runs-on: ${{ matrix.os }}   # ← OSを切り替え
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest, macos-latest]   # 複数プラットフォーム
        node: [16, 18, 20]                                  # 複数バージョン
    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: ${{ matrix.node }}

      - name: Install dependencies
        run: npm install

      - name: Run tests
        run: npm test
```

ランナーにNode.jsをセットアップする

GitHub Actionsによる継続的インテグレーション（CI）の例

```yaml
name: Matrix Example

on: [push]

jobs:
  test:
    runs-on: ${{ matrix.os }}   # ← OSを切り替え
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest, macos-latest]   # 複数プラットフォーム
        node: [16, 18, 20]                                  # 複数バージョン
    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: ${{ matrix.node }}

      - name: Install dependencies
        run: npm install

      - name: Run tests
        run: npm test

```

セットアップされるNode.jsのバージョンは、16, 18, 20 のいずれかとなる

GitHub Actionsによる継続的インテグレーション（CI）の例

```yaml
name: Matrix Example

on: [push]

jobs:
  test:
    runs-on: ${{ matrix.os }}   # ← OSを切り替え
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest, macos-latest]   # 複数プラットフォーム
        node: [16, 18, 20]                                  # 複数バージョン
    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: ${{ matrix.node }}

      - name: Install dependencies
        run: npm install

      - name: Run tests
        run: npm test
```

npm install を実行し、依存関係（このコードが使用するライブラリ）をインストールする

GitHub Actionsによる継続的インテグレーション（CI）の例

```yaml
name: Matrix Example

on: [push]

jobs:
  test:
    runs-on: ${{ matrix.os }}   # ← OSを切り替え
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest, macos-latest]   # 複数プラットフォーム
        node: [16, 18, 20]                                  # 複数バージョン
    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: ${{ matrix.node }}

      - name: Install dependencies
        run: npm install

      - name: Run tests
        run: npm test
```

最後にtestを実行する

モジュール2

- 継続的インテグレーションとは？
- GitHub Actionsによる継続的インテグレーションの例
- まとめ

まとめ

- **継続的インテグレーション**（Continuous Integration, CI）は、開発者がコードを頻繁に共有リポジトリへ統合するプロセスであり、自動でビルドやテストを実行する。
  - またオプションで文法やスタイルのチェック、セキュリティのチェックなども行う。
- CIにより、コードの不整合や不具合を早期に発見でき、コードの品質を保ち、開発スピードを上げることができる。
- GitHubリポジトリとGitHub Actionsを使用して、CIを実現できる
- **matrixストラテジー**を利用すると、複数のOS環境・複数の言語ランタイムバージョンの組み合わせの数だけジョブを生成し、並列でビルド・テストを実行できる。

目次

- ラーニングパス1
  - モジュール1 GitHub Actions を使用して開発タスクを自動化する
  - モジュール2 GitHub Actions を使用して継続的インテグレーション ワークフローを構築する
  - モジュール3 GitHub Actions を使ったアプリケーションのビルドと Azure へのデプロイ
  - モジュール4 GitHub スクリプトを使用した GitHub の自動化
- ラーニングパス2
  - モジュール5 GitHub Actions を活用して GitHub Packages に公開する
  - モジュール6 カスタム GitHub アクションを作成して公開する
  - モジュール7 企業で GitHub Actions を管理する

モジュール3

- Azure App Serviceとは？
- 「App Serviceプラン」と「App Serviceアプリ」とは？
- 運用スロットとは？
- ステージングスロットとは？
- デプロイとは？
- Azure App ServiceにWebアプリをデプロイするアクション
- まとめ

モジュール3

- Azure App Serviceとは？
- 「App Serviceプラン」と「App Serviceアプリ」とは？
- 運用スロットとは？
- ステージングスロットとは？
- デプロイとは？
- Azure App ServiceにWebアプリをデプロイするアクション
- まとめ

Azure App Serviceとは？

Azure App Serviceとは？

- Azureのサービスの一つ
- Webアプリやコンテナーアプリをを Azure上でホスティング
- たとえばPythonやJavaScript、C#などで開発されたWebアプリや、それらをコンテナー化したアプリなどをすばやくデプロイして運用できる
- PaaS（Platform as a Service）であり、運用の手間がかからない

Azure App ServiceはPaaS（Platform as a Service）であり、
IaaS（Infrastructure as a Service）に比べて、より多くの機能が提供される。

Azure仮想マシン
IaaS

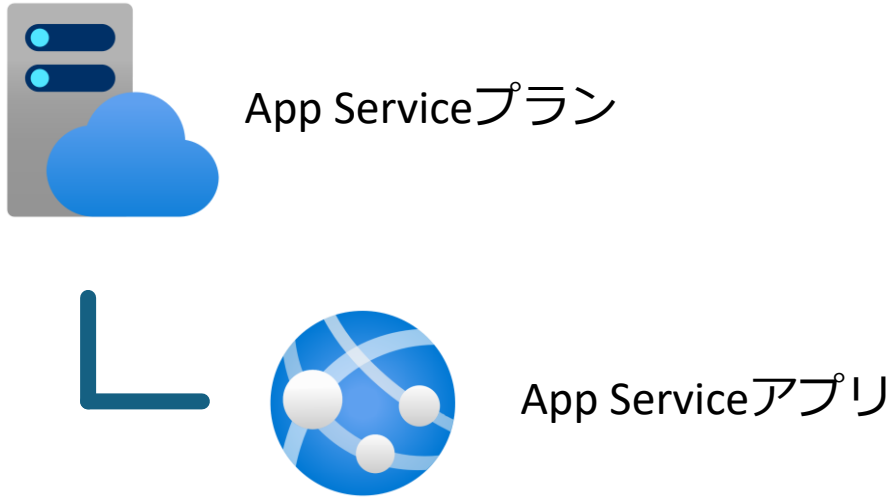
Azure App Service
PaaS

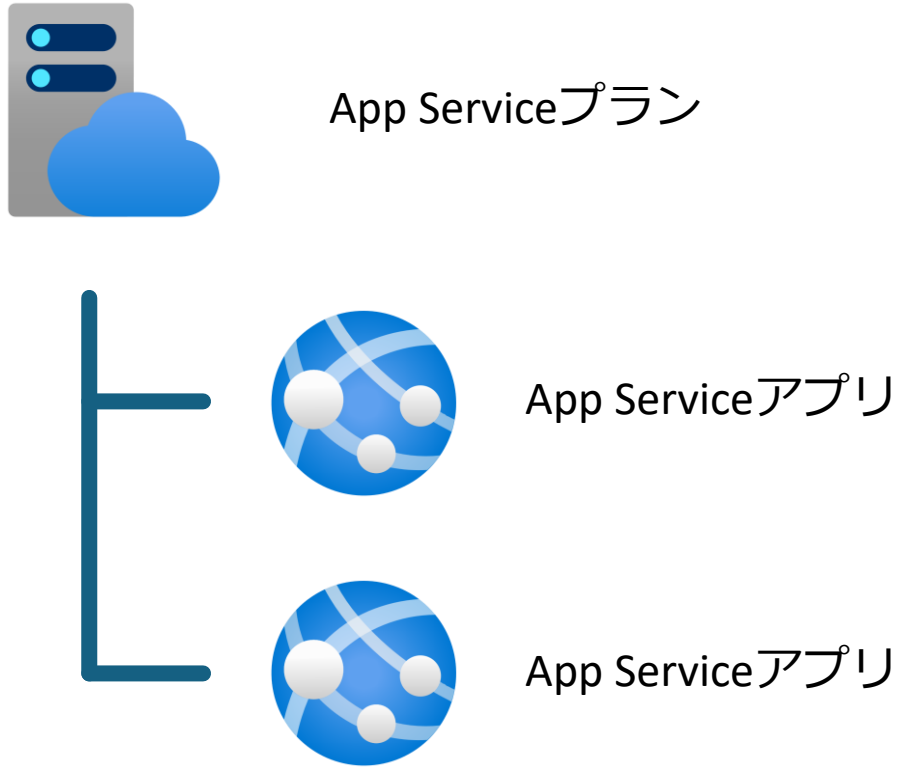
OSの管理：必要	OSの管理：不要
言語ランタイム：インストール必要	言語ランタイム：インストール不要 .NET/Java/Python/JavaScriptなど
スケーリングと負荷分散：運用が必要	スケーリングと負荷分散：組み込み
バックアップ：運用が必要	バックアップ：組み込み

モジュール3

- Azure App Serviceとは？
- 「App Serviceプラン」と「App Serviceアプリ」とは？
- 運用スロットとは？
- ステージングスロットとは？
- デプロイとは？
- Azure App ServiceにWebアプリをデプロイするアクション
- まとめ

Azure App Serviceでは、「App Serviceプラン」と「App Serviceアプリ」というリソースを使用して運用する。
1つのプランでは複数のアプリを運用できる。料金はアプリではなくプランに対して発生する。





App Serviceプランを作成する際に「価格レベル」を選択。
これにより、性能や、使用できる機能が変化し、料金も変わる。

価格レベル:
Basic

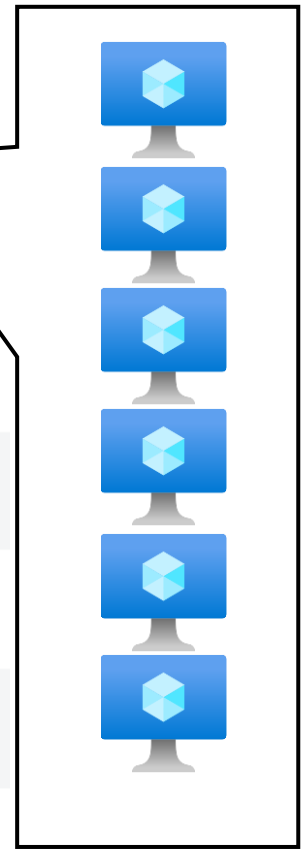
価格レベル:
Standard

価格レベル:
Premium

最大インスタンス数	最大 3	最大 10	最大 30*
カスタム ドメイン	サポート対象	サポート対象	サポート対象
自動スケール	–	サポート対象	サポート対象
ハイブリッド接続	サポート対象	サポート対象	サポート対象
仮想ネットワーク接続	サポート対象	サポート対象	サポート対象

https://learn.microsoft.com/ja-jp/azure/app-service/overview-hosting-plans

プラン内には最低1つの「インスタンス」が必要。インスタンスを増やすと、内部のWebアプリがよりたくさんのトラフィックを処理できるようになるが、よりコストもかかる。
Standard以上のプランでは自動スケールも利用可能。



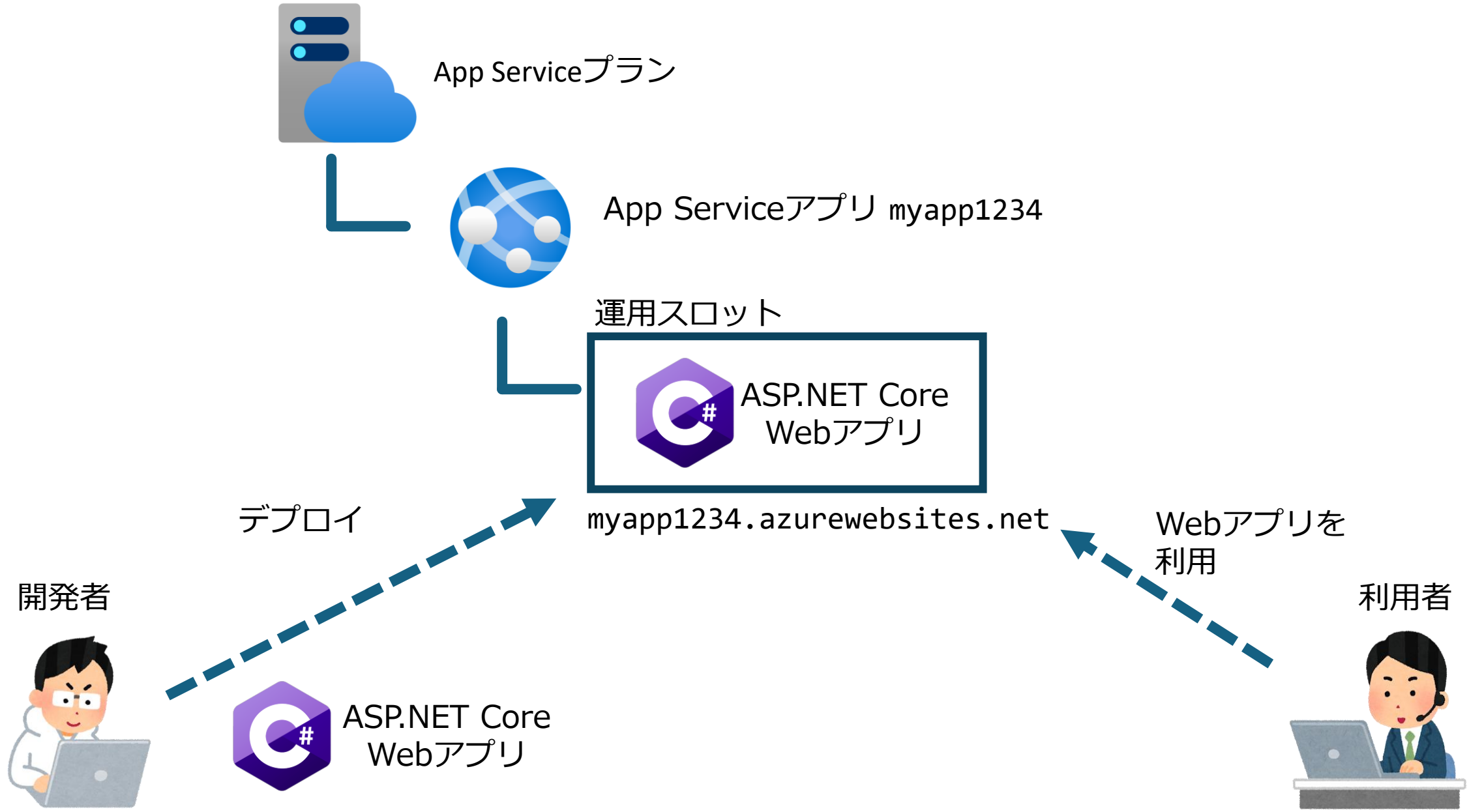
	価格レベル: Basic	価格レベル: Standard	価格レベル: Premium
最大インスタンス数	最大 3	最大 10	最大 30*
カスタム ドメイン	サポート対象	サポート対象	サポート対象
自動スケール	–	サポート対象	サポート対象
ハイブリッド接続	サポート対象	サポート対象	サポート対象
仮想ネットワーク接続	サポート対象	サポート対象	サポート対象

https://learn.microsoft.com/ja-jp/azure/app-service/overview-hosting-plans

モジュール3

- Azure App Serviceとは？
- 「App Serviceプラン」と「App Serviceアプリ」とは？
- 運用スロットとは？
- ステージングスロットとは？
- デプロイとは？
- Azure App ServiceにWebアプリをデプロイするアクション
- まとめ

アプリには「運用スロット」と呼ばれる場所があり、そこにアプリをデプロイする



モジュール3

- Azure App Serviceとは？
- 「App Serviceプラン」と「App Serviceアプリ」とは？
- 運用スロットとは？
- ステージングスロットとは？
- デプロイとは？
- Azure App ServiceにWebアプリをデプロイするアクション
- まとめ

ステージングスロットの利用

App Serviceアプリ myapp1234

Webアプリ(v1)
を利用

利用者

運用スロット

ASP.NET Core
Webアプリ
ver1

myapp1234.azurewebsites.net

ステージングスロット staging

ASP.NET Core
Webアプリ
ver2

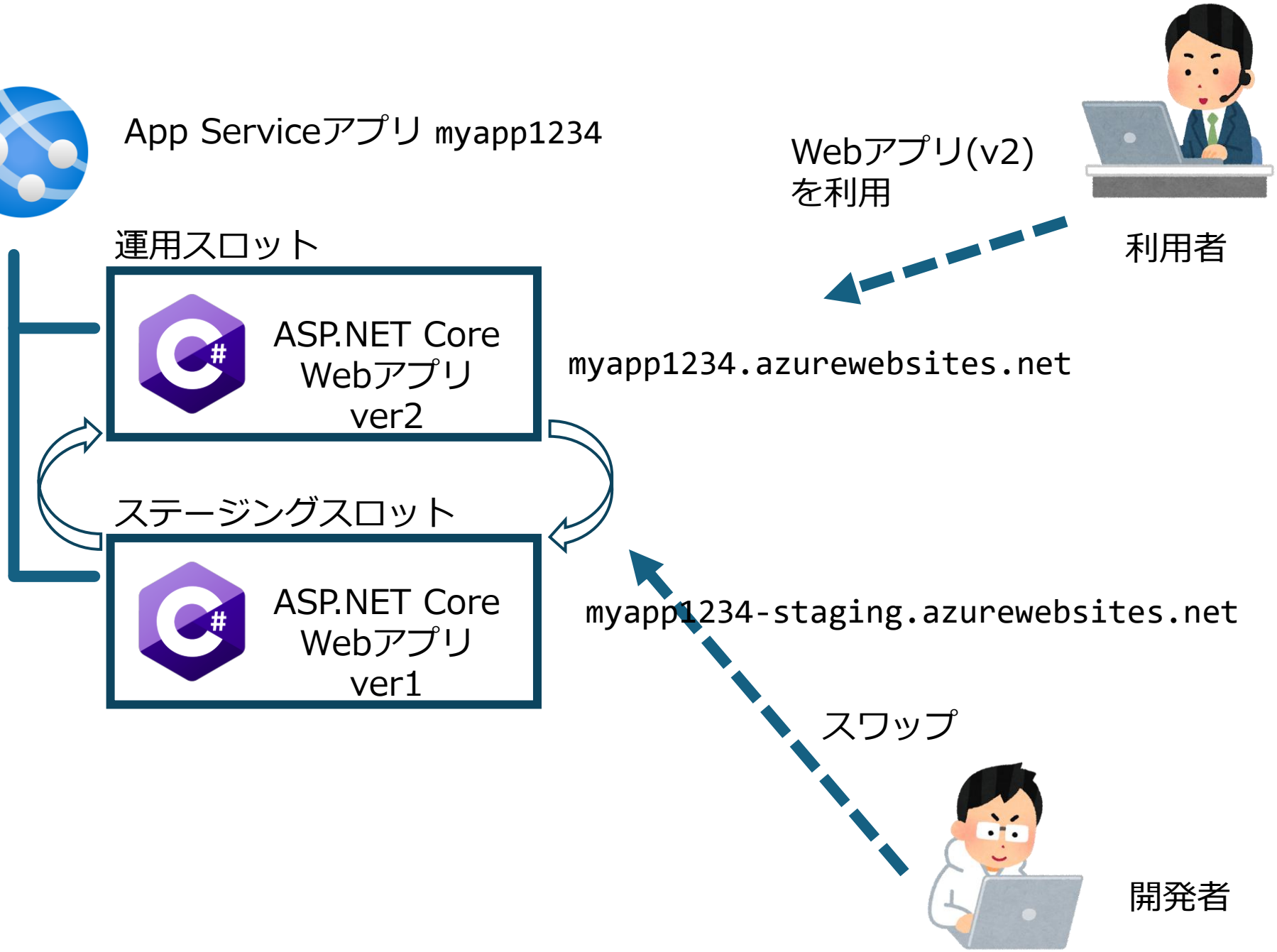
myapp1234-staging.azurewebsites.net

デプロイ

ASP.NET Core
Webアプリ
ver2

開発者

スワップを実行


App Serviceアプリ myapp1234
Webアプリ(v2)
を利用
利用者
運用スロット
ASP.NET Core
Webアプリ
ver2
myapp1234.azurewebsites.net
ステージングスロット
ASP.NET Core
Webアプリ
ver1
myapp1234-staging.azurewebsites.net
スワップ
開発者

モジュール3

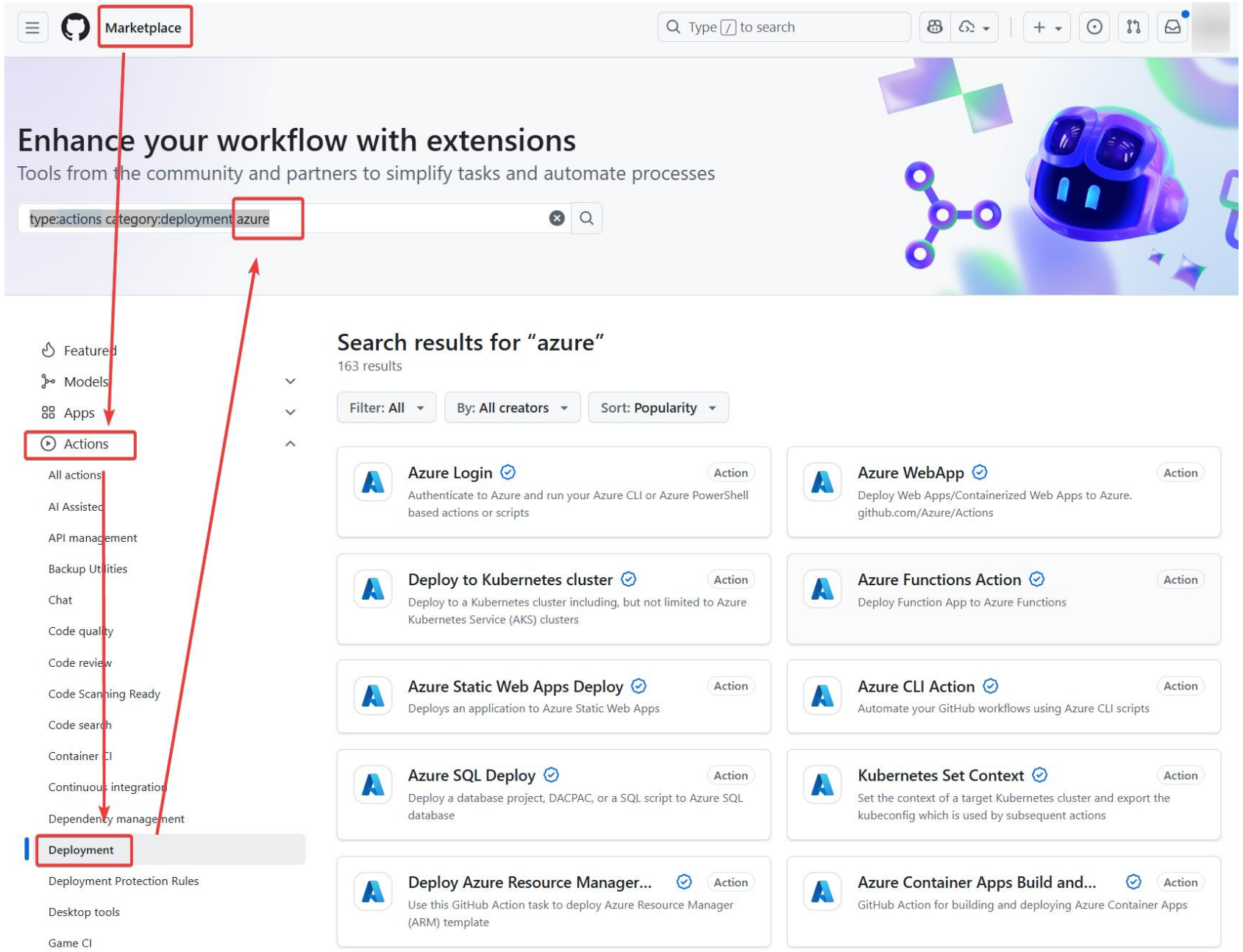
- Azure App Serviceとは？
- 「App Serviceプラン」と「App Serviceアプリ」とは？
- 運用スロットとは？
- ステージングスロットとは？
- デプロイとは？
- Azure App ServiceにWebアプリをデプロイするアクション
- まとめ

デプロイとは？

- ビルドされたコードをテスト環境・ステージング環境・本番環境などに配置して、その環境でコードを動かせるようにすること
- たとえばAzure App ServiceにWebアプリをデプロイするなど。

- GitHub Actionsには、デプロイ用のアクションが多数用意されている

Marketplaceでデプロイ用のアクションを検索し、さらにAzureと検索キーワードを追加すると、Azure関連の多数のデプロイアクションが見つかる



モジュール3

- Azure App Serviceとは？
- 「App Serviceプラン」と「App Serviceアプリ」とは？
- 運用スロットとは？
- ステージングスロットとは？
- デプロイとは？
- Azure App ServiceにWebアプリをデプロイするアクション
- まとめ

Azure App ServiceにWebアプリをデプロイするアクション

- azure/webapps-deploy

- GitHub Marketplace: https://github.com/marketplace/actions/azure-webapp

- GitHubリポジトリ: https://github.com/Azure/webapps-deploy

モジュール3

- Azure App Serviceとは？
- 「App Serviceプラン」と「App Serviceアプリ」とは？
- 運用スロットとは？
- ステージングスロットとは？
- デプロイとは？
- Azure App ServiceにWebアプリをデプロイするアクション
- まとめ

まとめ

- **Azure App Service**は、Azureのサービスの一つであり、Webアプリやコンテナーアプリをazure上でホスティングする。
  - PythonやJavaScript、C#などで開発されたWebアプリや、それらをコンテナー化したアプリなどをすばやくデプロイして運用できる。
  - PaaS（Platform as a Service）であり、運用の手間がかからない

- Azure App ServiceにWebアプリをデプロイするアクション「**azure/azure-webapp**」を利用すると、GitHubリポジトリのWebアプリのコードをAzure App Serviceへとすばやくデプロイできる。

目次

- ラーニングパス1
  - モジュール1 GitHub Actions を使用して開発タスクを自動化する
  - モジュール2 GitHub Actions を使用して継続的インテグレーション ワークフローを構築する
  - モジュール3 GitHub Actions を使ったアプリケーションのビルドと Azure へのデプロイ
  - モジュール4 GitHub スクリプトを使用した GitHub の自動化
- ラーニングパス2
  - モジュール5 GitHub Actions を活用して GitHub Packages に公開する
  - モジュール6 カスタム GitHub アクションを作成して公開する
  - モジュール7 企業で GitHub Actions を管理する

モジュール4

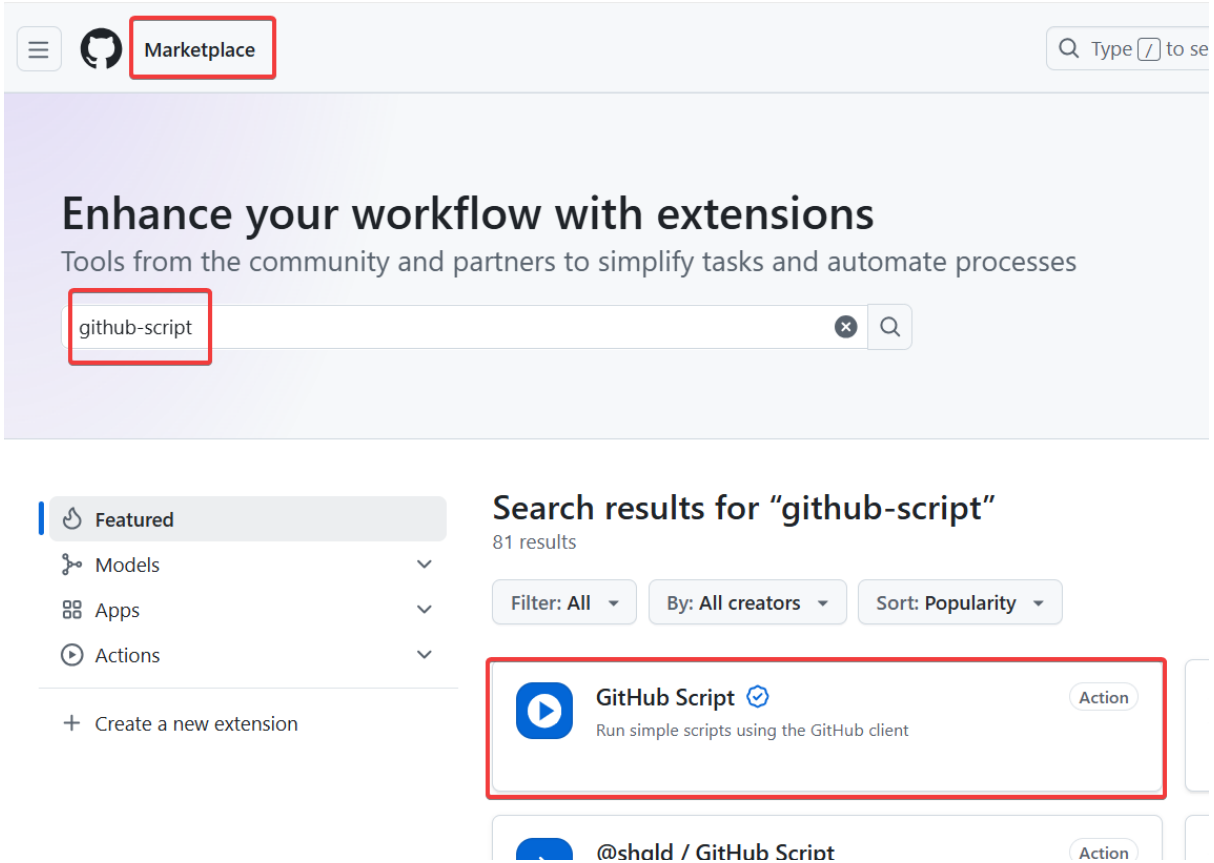
- github-script とは？
- github-scriptを使ったワークフローの例
- 実際の開発におけるgithub-scriptの活用例
- まとめ

モジュール4

- github-script とは？
- github-scriptを使ったワークフローの例
- 実際の開発におけるgithub-scriptの活用例
- まとめ

github-script とは？

- GitHub Actionsのアクションの一つ
- アクションとして任意のJavaScriptを実行できる
- JavaScriptを使用して、GitHubの各種操作を実行できる
  - Issueにコメントする
  - Issueにラベルを付与する



モジュール4

- github-script とは？
- github-scriptを使ったワークフローの例
- 実際の開発におけるgithub-scriptの活用例
- まとめ

ワークフローの例

main　repo20 / .github / workflows / kaomoji.yml

.github/workflows/kaomoji.yml

hiryamada Update kaomoji.yml

Code　Blame

```yaml
name: Add Kaomoji to Issue Comments

on:
  issue_comment:
    types: [created]

permissions:
  issues: write

jobs:
  add-kaomoji:
    runs-on: ubuntu-latest
    steps:
      - name: Add Kaomoji
        uses: actions/github-script@v7
        with:
          script: |
            const kaomojis = ["( ^ ▽ ^ )", "(ﾉ✧▽✧)ﾉ", "ヾ(´ᗜ`*)ﾉ", "(•ᴗ•)ﾉ"];
            const random = kaomojis[Math.floor(Math.random() * kaomojis.length)];
            github.rest.issues.createComment({
              issue_number: context.issue.number,
              owner: context.repo.owner,
              repo: context.repo.repo,
              body: `${random}`
            });
```

YAML形式で
ワークフローを定義

ワークフローの例

```yaml
name: Add Kaomoji to Issue Comments

on:
  issue_comment:
    types: [created]

permissions:
  issues: write

jobs:
  add-kaomoji:
    runs-on: ubuntu-latest
    steps:
      - name: Add Kaomoji
        uses: actions/github-script@v7
        with:
          script: |
            const kaomojis = ["( ^ ▽ ^ )", "(ノ ✧▽✧)ノ ", "٩( ´ ᗜ ` *)و", "(•••)و"];
            const random = kaomojis[Math.floor(Math.random() * kaomojis.length)];
            github.rest.issues.createComment({
              issue_number: context.issue.number,
              owner: context.repo.owner,
              repo: context.repo.repo,
              body: `${random}`
            });
```

ワークフローの
わかりやすい表示名

ワークフローの例

```yaml
name: Add Kaomoji to Issue Comments

on:
  issue_comment:
    types: [created]

permissions:
  issues: write

jobs:
  add-kaomoji:
    runs-on: ubuntu-latest
    steps:
      - name: Add Kaomoji
        uses: actions/github-script@v7
        with:
          script: |
            const kaomojis = ["( ^ ▽ ^ )", "(ノ ✧▽✧)ノ ", "٩( ´ ᗜ ` *)و", "(•••)و"];
            const random = kaomojis[Math.floor(Math.random() * kaomojis.length)];
            github.rest.issues.createComment({
              issue_number: context.issue.number,
              owner: context.repo.owner,
              repo: context.repo.repo,
              body: `${random}`
            });
```

このワークフローを動かす「イベント」の指定。
この例では「Issueのコメントが作成されたとき」と指定している。

この部分はトリガーとも呼ばれる

ワークフローの例

```yaml
name: Add Kaomoji to Issue Comments

on:
  issue_comment:
    types: [created]

permissions:
  issues: write

jobs:
  add-kaomoji:
    runs-on: ubuntu-latest
    steps:
      - name: Add Kaomoji
        uses: actions/github-script@v7
        with:
          script: |
            const kaomojis = ["( ^ ▽ ^ )", "(ノ ✧▽✧)ノ ", "٩( ´ ᗜ ` *)و", "(•••)و"];
            const random = kaomojis[Math.floor(Math.random() * kaomojis.length)];
            github.rest.issues.createComment({
              issue_number: context.issue.number,
              owner: context.repo.owner,
              repo: context.repo.repo,
              body: `${random}`
            });
```

このワークフローに対し
Issueへの書き込み権限を与える

ワークフローの例

```yaml
name: Add Kaomoji to Issue Comments

on:
  issue_comment:
    types: [created]

permissions:
  issues: write

jobs:
  add-kaomoji:
    runs-on: ubuntu-latest
    steps:
      - name: Add Kaomoji
        uses: actions/github-script@v7
        with:
          script: |
            const kaomojis = ["( ^ ▽ ^ )", "(ノ ✧▽✧)ノ ", "٩( ´ ᗝ ` *)و", "(•••)و"];
            const random = kaomojis[Math.floor(Math.random() * kaomojis.length)];
            github.rest.issues.createComment({
              issue_number: context.issue.number,
              owner: context.repo.owner,
              repo: context.repo.repo,
              body: `${random}`
            });
```

このワークフローで実行する
「ジョブ」を指定

ワークフローの例

```yaml
name: Add Kaomoji to Issue Comments

on:
  issue_comment:
    types: [created]

permissions:
  issues: write

jobs:
  add-kaomoji:
    runs-on: ubuntu-latest
    steps:
      - name: Add Kaomoji
        uses: actions/github-script@v7
        with:
          script: |
            const kaomojis = ["( ^ ▽ ^ )", "(ノ ✧▽✧)ノ ", "٩( ´ ᗜ ` *)و", "(•••)و"];
            const random = kaomojis[Math.floor(Math.random() * kaomojis.length)];
            github.rest.issues.createComment({
              issue_number: context.issue.number,
              owner: context.repo.owner,
              repo: context.repo.repo,
              body: `${random}`
            });
```

「ジョブ」の中で動かす
「ステップ」の指定。

ワークフローの例

```yaml
name: Add Kaomoji to Issue Comments

on:
  issue_comment:
    types: [created]

permissions:
  issues: write

jobs:
  add-kaomoji:
    runs-on: ubuntu-latest
    steps:
      - name: Add Kaomoji
        uses: actions/github-script@v7
        with:
          script: |
            const kaomojis = ["( ^ ▽ ^ )", "(ノ ✧▽✧)ノ ", "٩( ´ ᗜ ` *)و", "(•••)و"];
            const random = kaomojis[Math.floor(Math.random() * kaomojis.length)];
            github.rest.issues.createComment({
              issue_number: context.issue.number,
              owner: context.repo.owner,
              repo: context.repo.repo,
              body: `${random}`
            });
```

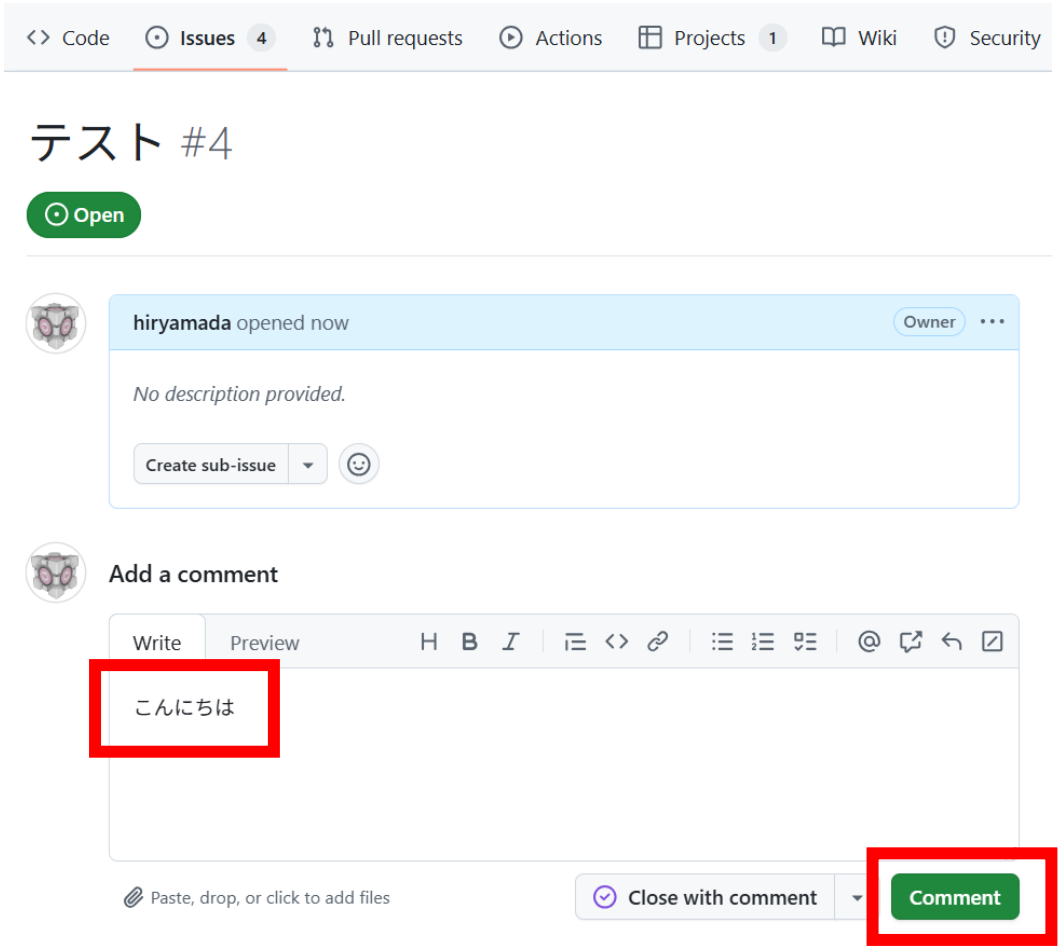
この「ステップ」では
github-scriptアクションを利用し、
JavaScriptを実行

ワークフローの例

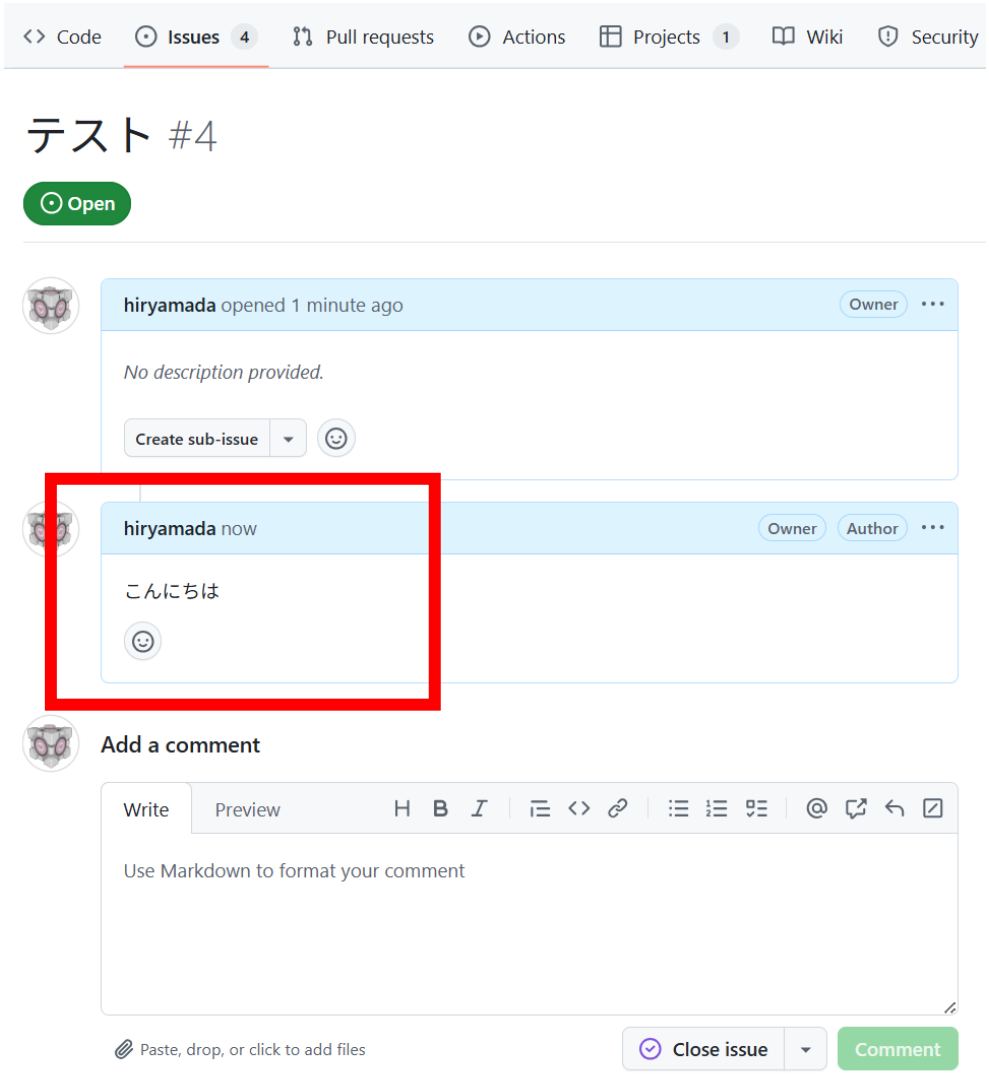
```yaml
name: Add Kaomoji to Issue Comments

on:
  issue_comment:
    types: [created]

permissions:
  issues: write

jobs:
  add-kaomoji:
    runs-on: ubuntu-latest
    steps:
      - name: Add Kaomoji
        uses: actions/github-script@v7
        with:
          script: |
            const kaomojis = ["( ^ ▽ ^ )", "(ノ ✧▽✧)ノ ", "٩( ' ᗝ ` *)و", "(•••)و"];
            const random = kaomojis[Math.floor(Math.random() * kaomojis.length)];
            github.rest.issues.createComment({
              issue_number: context.issue.number,
              owner: context.repo.owner,
              repo: context.repo.repo,
              body: `${random}`
            });
```

JavaScriptを使用して
Issueにランダムなコメントを追加

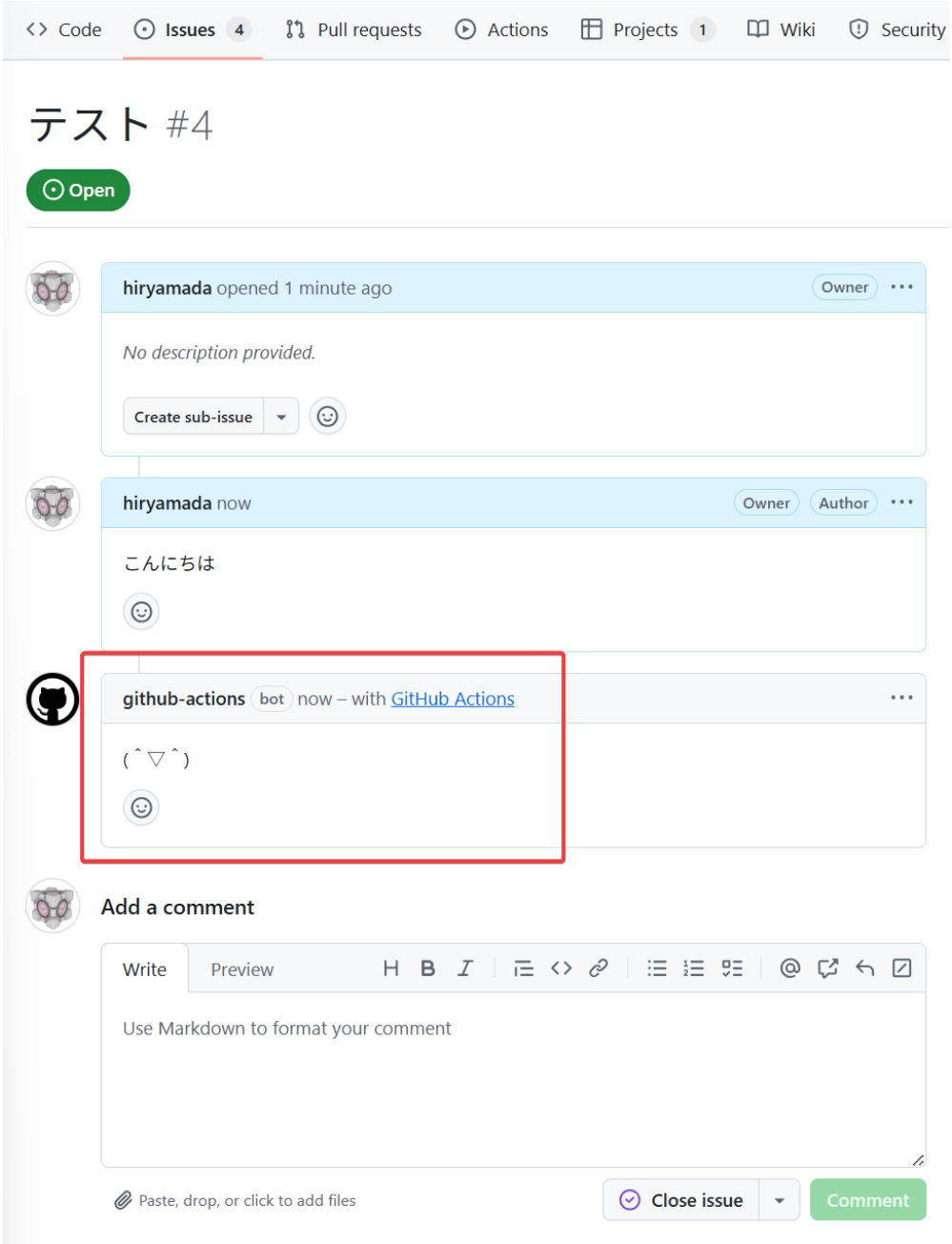
Issueにテストのコメントを書き込む



コメントが書き込まれた。するとここで**GitHub Actionsのワークフロー**が動き出す



ワークフローにより、自動的に新しいコメントが付けられた！



モジュール4

- github-script とは？
- github-scriptを使ったワークフローの例
- 実際の開発におけるgithub-scriptの活用例
- まとめ

実際の開発におけるgithub-script（コメント自動付与）の活用例

- テストが失敗した際に自動的にコメントで通知する
- 作成されてから長期間放置されているIssueに対し、対処を促すコメントを自動追加する
- コードがプッシュされた際にコードをチェックし、コーディング規約違反やセキュリティ脆弱性などの問題が見つかった場合はIssueにコメントを自動追加する
- サーバーの状態を調べ、問題があればIssueのコメントを自動追加する

【GitHub】Actionsを使ってコミット時のコードチェック・ユニットテストの自動化をする

非アクティブな Issue をクローズする - GitHub ドキュメント

GitHub Actions で失敗したテストをコメントで通知する Action の作成 - スタディサプリ Product Team Blog

モジュール4

- github-script とは？
- github-scriptを使ったワークフローの例
- 実際の開発におけるgithub-scriptの活用例
- まとめ

まとめ

- **actions/github-script**は、GitHub Actionsのアクションの一つであり、任意のJavaScriptを実行できる。

- JavaScriptを使用して、Issueにコメントを付与する、Issueにラベルを付与するといった、GitHubの各種操作を自動化できる。

目次

- ラーニングパス1
  - モジュール1 GitHub Actions を使用して開発タスクを自動化する
  - モジュール2 GitHub Actions を使用して継続的インテグレーション ワークフローを構築する
  - モジュール3 GitHub Actions を使ったアプリケーションのビルドと Azure へのデプロイ
  - モジュール4 GitHub スクリプトを使用した GitHub の自動化
- ラーニングパス2
  - モジュール5 GitHub Actions を活用して GitHub Packages に公開する
  - モジュール6 カスタム GitHub アクションを作成して公開する
  - モジュール7 企業で GitHub Actions を管理する

モジュール5

- GitHub Packagesとは？
- GitHub Packagesのメリット
- パッケージの可視性
- Dockerとは？
- GitHub Container Registry(GHCR)とは？
- コンテナーをビルドしGHCRに格納するワークフローの例
- 参考: GITHUB_TOKENとは？
- まとめ

モジュール5

- GitHub Packagesとは？
- GitHub Packagesのメリット
- パッケージの可視性
- Dockerとは？
- GitHub Container Registry(GHCR)とは？
- コンテナーをビルドしGHCRに格納するワークフローの例
- 参考: GITHUB_TOKENとは？
- まとめ

GitHub Packagesとは？

- GitHubが提供する**パッケージ**の管理・ホスティングサービス
  - **パッケージ**とは、ソフトウェアやライブラリをまとめて配布・再利用できるようにしたもののこと
  - 具体的にはReact、 Express、 Vue.js、 NumPy、 Pandas、 TensorFlow、 Spring Framework、 Hibernate、 Newtonsoft.Json、 Entity Frameworkなどのパッケージがある
- GitHub Packagesは、 npm、 RubyGems、 Apache Maven、 Gradle、 Docker、 NuGetといったパッケージマネージャーに対応する**レジストリ**を提供
  - **レジストリ**とは、パッケージを保存・公開・検索できる場所のこと
- Dockerコンテナーを格納することもできる（GitHub Container Registry、 GHCR）

モジュール5

- GitHub Packagesとは？
- GitHub Packagesのメリット
- パッケージの可視性
- Dockerとは？
- GitHub Container Registry(GHCR)とは？
- コンテナーをビルドしGHCRに格納するワークフローの例
- 参考: GITHUB_TOKENとは？
- まとめ

GitHub Packagesのメリット

- 自作のソフトウェアのパッケージを作成し、GitHub Packagesのパブリックリポジトリに格納すれば、世界中の開発者がそのパッケージを利用できる
  - たとえば開発者は npm install @USERNAME/PACKAGENAME といった形でそのパッケージを利用できる

- GitHubやGitHub Actionsと統合されており、パッケージの作成とリポジトリへの格納が容易

モジュール5

- GitHub Packagesとは？
- GitHub Packagesのメリット
- パッケージの可視性
- Dockerとは？
- GitHub Container Registry(GHCR)とは？
- コンテナーをビルドしGHCRに格納するワークフローの例
- 参考: GITHUB_TOKENとは？
- まとめ

パッケージの可視性

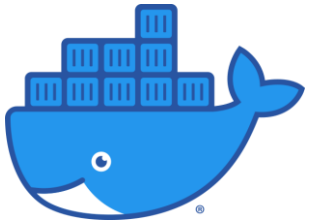
- GitHubパブリックリポジトリでパッケージを公開（publish）した場合、パッケージは世界中の開発者が利用できる

- GitHubプライベートリポジトリでパッケージを公開した場合、パッケージリポジトリのコラボレータ（共同開発者）や、GitHub組織（GitHub Organizations）内のユーザーが利用できる

モジュール5

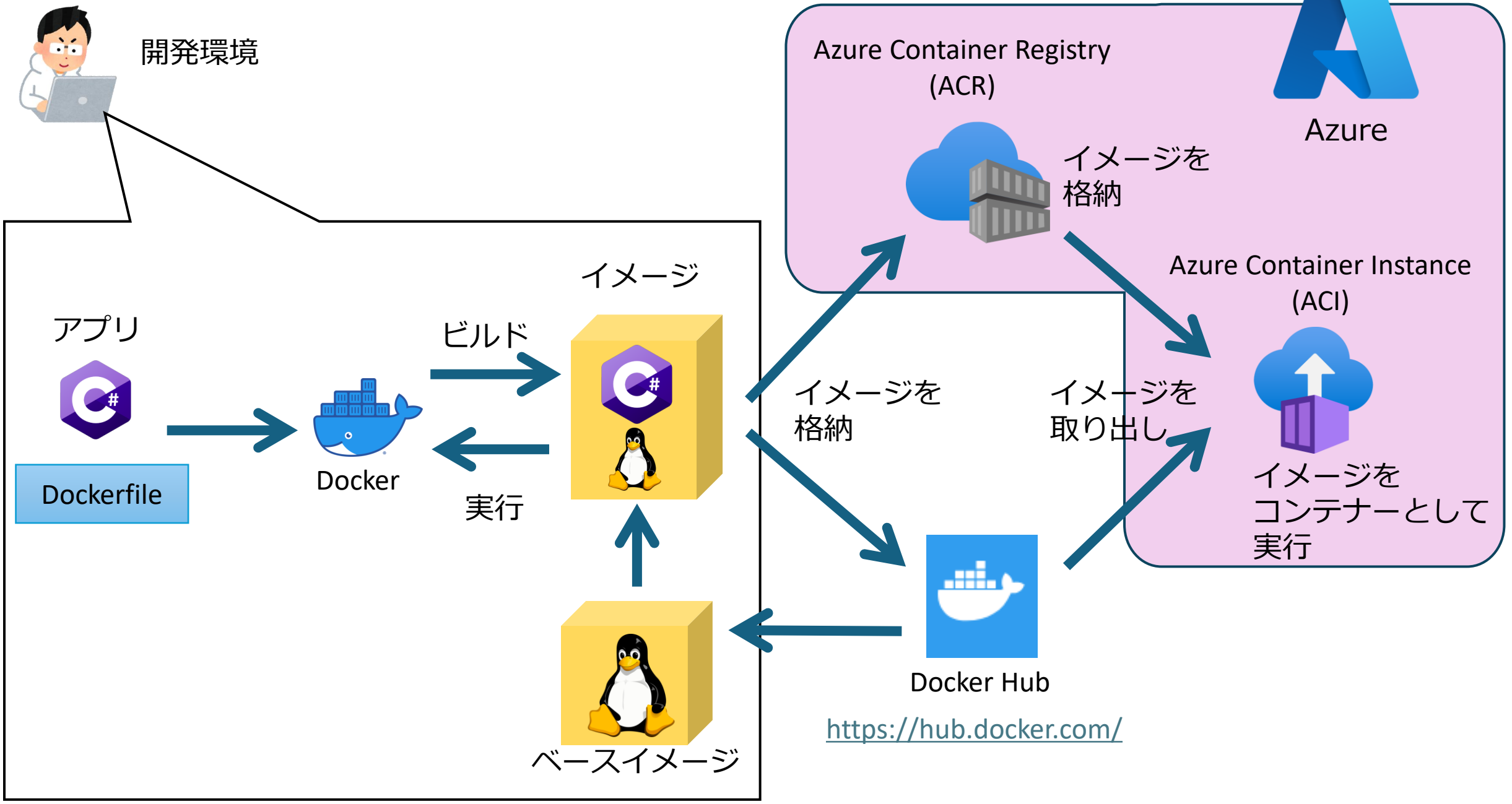
- GitHub Packagesとは？
- GitHub Packagesのメリット
- パッケージの可視性
- Dockerとは？
- GitHub Container Registry(GHCR)とは？
- コンテナーをビルドしGHCRに格納するワークフローの例
- 参考: GITHUB_TOKENとは？
- まとめ

Dockerとは？



- コンテナー型仮想化ツールの一種
- ソフトウェアの実行に必要なコード、ランタイム、設定ファイル、ライブラリなどを**コンテナー**としてパッケージ化
  - 再利用がしやすい
- コンテナーにはosが含まれない
  - コンテナーは従来の「サーバー仮想化」よりも軽量（サイズが小さい）
- 「コンテナー」は開発環境・本番環境、オンプレミスサーバー・クラウドなどどこでも同じように動くことが期待できる
  - 可搬性（ポータビリティ）が高い

コンテナーの開発と実行の例



モジュール5

- GitHub Packagesとは？
- GitHub Packagesのメリット
- パッケージの可視性
- Dockerとは？
- GitHub Container Registry(GHCR)とは？
- コンテナーをビルドしGHCRに格納するワークフローの例
- 参考: GITHUB_TOKENとは？
- まとめ

GitHub Container Registry(GHCR)とは？

- Dockerコンテナーのイメージを格納するレジストリを提供するサービス
- GitHubリポジトリやGitHub Actionsと統合されており、使いやすい
  - ソースコードの管理はGitHubリポジトリ
  - CI/CDによるイメージのビルドとデプロイはGitHub Actions
  - イメージの管理はGHCR
- レジストリのURLは `ghcr.io`

モジュール5

- GitHub Packagesとは？
- GitHub Packagesのメリット
- パッケージの可視性
- Dockerとは？
- GitHub Container Registry(GHCR)とは？
- コンテナーをビルドしGHCRに格納するワークフローの例
- 参考: GITHUB_TOKENとは？
- まとめ

コンテナーをビルドし、GHCRに格納するワークフローの例（主要部分の抜粋）

```yaml
steps:
  - name: Checkout repository
    uses: actions/checkout@v5

  - name: Log in to the Container registry
    uses: docker/login-action@v3
    with:
      registry: ${{ env.REGISTRY }}
      username: ${{ github.actor }}
      password: ${{ secrets.GITHUB_TOKEN }}

  - name: Extract metadata (tags, labels) for Docker
    id: meta
    uses: docker/metadata-action@v5
    with:
      images: ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}

  - name: Build and push Docker image
    id: push
    uses: docker/build-push-action@v6
    with:
      context: .
      push: true
      tags: ${{ steps.meta.outputs.tags }}
      labels: ${{ steps.meta.outputs.labels }}
```

リポジトリから
コードを取り出し

コンテナーをビルドし、GHCRに格納するワークフローの例（主要部分の抜粋）

```yaml
steps:
  - name: Checkout repository
    uses: actions/checkout@v5

  - name: Log in to the Container registry
    uses: docker/login-action@v3
    with:
      registry: ${{ env.REGISTRY }}
      username: ${{ github.actor }}
      password: ${{ secrets.GITHUB_TOKEN }}

  - name: Extract metadata (tags, labels) for Docker
    id: meta
    uses: docker/metadata-action@v5
    with:
      images: ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}

  - name: Build and push Docker image
    id: push
    uses: docker/build-push-action@v6
    with:
      context: .
      push: true
      tags: ${{ steps.meta.outputs.tags }}
      labels: ${{ steps.meta.outputs.labels }}
```

GHCRにログイン

コンテナーをビルドし、GHCRに格納するワークフローの例（主要部分の抜粋）

```yaml
steps:
  - name: Checkout repository
    uses: actions/checkout@v5

  - name: Log in to the Container registry
    uses: docker/login-action@v3
    with:
      registry: ${{ env.REGISTRY }}
      username: ${{ github.actor }}
      password: ${{ secrets.GITHUB_TOKEN }}

  - name: Extract metadata (tags, labels) for Docker
    id: meta
    uses: docker/metadata-action@v5
    with:
      images: ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}

  - name: Build and push Docker image
    id: push
    uses: docker/build-push-action@v6
    with:
      context: .
      push: true
      tags: ${{ steps.meta.outputs.tags }}
      labels: ${{ steps.meta.outputs.labels }}
```

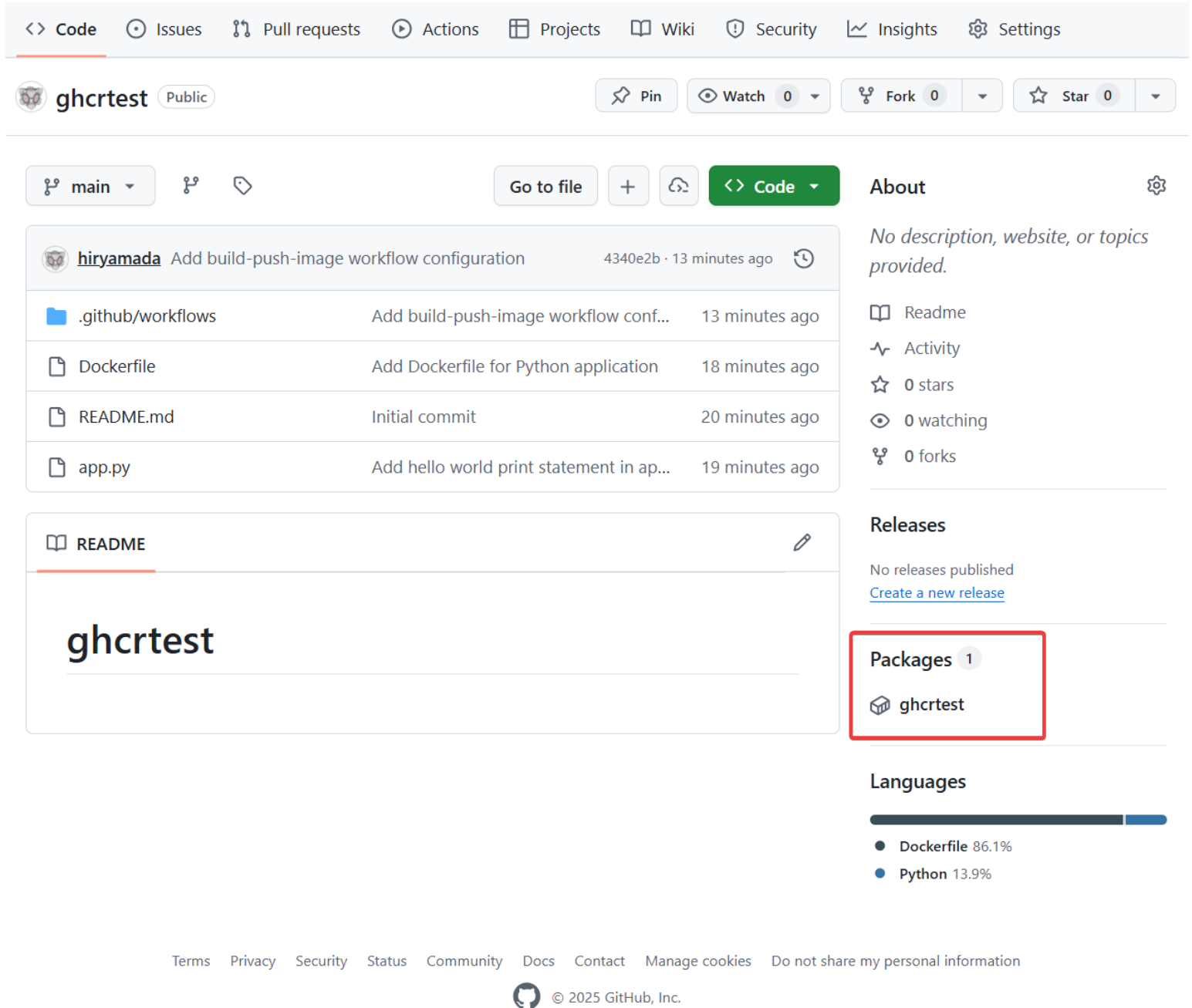
イメージに付与する
タグ、ラベルの
情報を抽出

コンテナーをビルドし、GHCRに格納するワークフローの例（主要部分の抜粋）

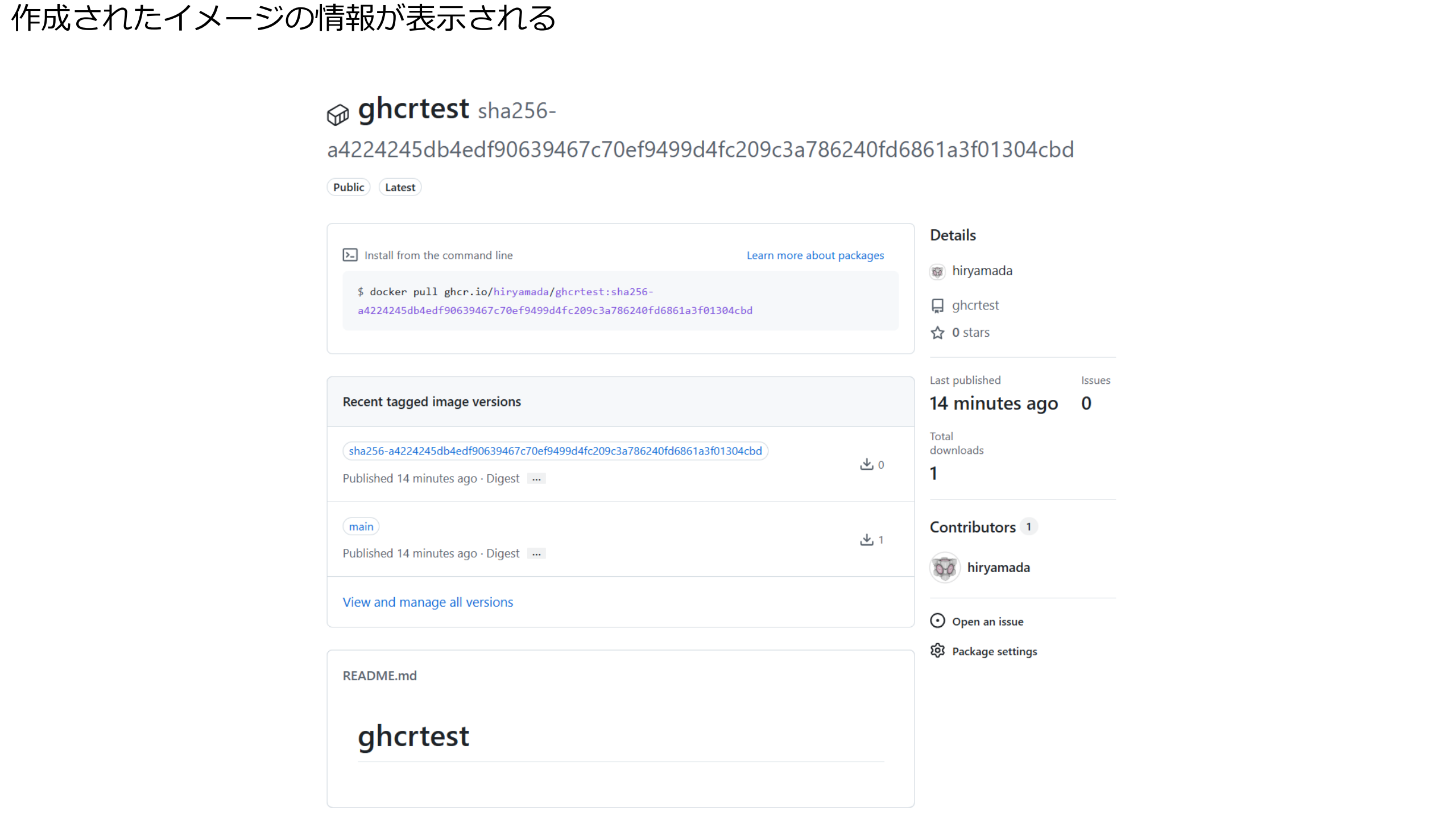
```yaml
steps:
  - name: Checkout repository
    uses: actions/checkout@v5

  - name: Log in to the Container registry
    uses: docker/login-action@v3
    with:
      registry: ${{ env.REGISTRY }}
      username: ${{ github.actor }}
      password: ${{ secrets.GITHUB_TOKEN }}

  - name: Extract metadata (tags, labels) for Docker
    id: meta
    uses: docker/metadata-action@v5
    with:
      images: ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}

  - name: Build and push Docker image
    id: push
    uses: docker/build-push-action@v6
    with:
      context: .
      push: true
      tags: ${{ steps.meta.outputs.tags }}
      labels: ${{ steps.meta.outputs.labels }}
```

イメージをビルドし
GHCRにプッシュ

ワークフローが実行されると、GitHubリポジトリの右下の「Packages」にパッケージが表示される



作成されたイメージの情報が表示される



モジュール5

- GitHub Packagesとは？
- GitHub Packagesのメリット
- パッケージの可視性
- Dockerとは？
- GitHub Container Registry(GHCR)とは？
- コンテナーをビルドしGHCRに格納するワークフローの例
- 参考: GITHUB_TOKENとは？
- まとめ

参考: GITHUB_TOKEN とは？

```yaml
steps:
  - name: Checkout repository
    uses: actions/checkout@v5

  - name: Log in to the Container registry
    uses: docker/login-action@v3
    with:
      registry: ${{ env.REGISTRY }}
      username: ${{ github.actor }}
      password: ${{ secrets.GITHUB_TOKEN }}

  - name: Extract metadata (tags, labels) for Docker
    id: meta
    uses: docker/metadata-action@v5
    with:
      images: ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}

  - name: Build and push Docker image
    id: push
    uses: docker/build-push-action@v6
    with:
      context: .
      push: true
      tags: ${{ steps.meta.outputs.tags }}
      labels: ${{ steps.meta.outputs.labels }}
```

GHCRにログインする際、
GITHUB_TOKENというもの
を
渡している

GITHUB_TOKEN とは？

- 各ワークフローのジョブ開始時に、GitHubは **GITHUB_TOKEN** を自動的に作成する
- これはリポジトリのチェックアウト（actions/checkout）、IssueやPull Requestの操作（コメントなど）、GitHub Packages・GitHub Container Registryへのログイン、GitHub APIの呼び出しなどで使用される**一時的な認証情報**
  - パスワードのようなもの
  - ジョブ終了時または最大24時間で失効する
- ユーザーが手動で発行する**Personal Access Token（PAT）**と異なり、GITHUB_TOKENは自動生成・短期利用されるため安全。

GITHUB_TOKEN - GitHub ドキュメント

モジュール5

- GitHub Packagesとは？
- GitHub Packagesのメリット
- パッケージの可視性
- Dockerとは？
- GitHub Container Registry(GHCR)とは？
- コンテナーをビルドしGHCRに格納するワークフローの例
- 参考: GITHUB_TOKENとは？
- まとめ

まとめ

- **パッケージ**とは、ソフトウェアやライブラリをまとめて配布・再利用できるようにしたもの。React、Express、Vue.js、NumPy、Pandas、TensorFlow、Spring Framework、Hibernate、Newtonsoft.Json、Entity Frameworkなどのソフトウェアが、パッケージの形で提供されている。
- **レジストリ**とは、パッケージを保存・公開・検索できる場所のこと。npm、RubyGems、Apache Maven、Gradle、Docker、NuGetといった**パッケージマネージャー**を使用して、レジストリにアクセスする。
- 自作のパッケージをレジストリに入れて公開することで、組織内、あるいは世界中の開発者に、パッケージを利用してもらうことができる。
- **GitHub Packages**は、GitHubが提供するパッケージの管理・ホスティングのサービス。
- **Docker**は、コンテナー型仮想化ツールの一種。
- **コンテナー**にはソフトウェアの実行に必要なコード、ランタイム、設定ファイル、ライブラリなどが含まれており、軽量でポータビリティが高い。
- GitHub Packagesの一部である**GitHub Container Registry（GHCR）**には、Dockerコンテナーのイメージを格納できる。

目次

- ラーニングパス1
  - モジュール1 GitHub Actions を使用して開発タスクを自動化する
  - モジュール2 GitHub Actions を使用して継続的インテグレーション ワークフローを構築する
  - モジュール3 GitHub Actions を使ったアプリケーションのビルドと Azure へのデプロイ
  - モジュール4 GitHub スクリプトを使用した GitHub の自動化
- ラーニングパス2
  - モジュール5 GitHub Actions を活用して GitHub Packages に公開する
  - モジュール6 カスタム GitHub アクションを作成して公開する
  - モジュール7 企業で GitHub Actions を管理する

モジュール6

- カスタムアクションとは？
- カスタムアクションの公開
- カスタムアクションの種類と構成例
- 複合アクションの利用例
- まとめ

モジュール6

- カスタムアクションとは？
- カスタムアクションの公開
- カスタムアクションの種類と構成例
- 複合アクションの利用例
- まとめ

カスタムアクションとは？

- GitHubのユーザーが必要に応じて作成・利用できる、再利用可能な処理の部品
  - 繰り返し記述される部分をくくりだして部品化できるので便利

- カスタムアクションはGitHubリポジトリ内の `.github/actions/(カスタムアクション名)/` 以下に配置される
- 例:
- `.github/actions/action-a/`... action-aのファイルを置く
- `.github/actions/action-b/` ... action-bのファイルを置く

モジュール6

- カスタムアクションとは？
- カスタムアクションの公開
- カスタムアクションの種類と構成例
- 複合アクションの利用例
- まとめ

カスタムアクションの公開

- GitHub Marketplaceに公開して、他のGitHubユーザーにカスタムアクションを使ってもらうことも可能

- 公開せず、プライベートリポジトリの中だけで利用することも可能

- 組織（企業）としてカスタムアクションを作成し、組織内で共同利用するということも可能

モジュール6

- カスタムアクションとは？
- カスタムアクションの公開
- カスタムアクションの種類と構成例
- 複合アクションの利用例
- まとめ

カスタムアクションの種類と構成例

- 複合アクション（Composite actions）
  - 複数のステップをひとつにまとめたアクション

```
.github/actions/action-a/
└ action.yml
```

- JavaScriptアクション
  - Node.jsで実行されるアクション

```
.github/actions/action-b/
├ index.js
├ package.json
└ action.yml
```

- Dockerコンテナーアクション
  - Dockerコンテナー内で実行されるアクション

```
.github/actions/action-c/
├ Dockerfile
├ entrypoint.sh
└ action.yml
```

カスタムアクションの種類と構成例

- 複合アクション（Composite actions）
  - 複数のステップをひとつにまとめたアクション

```
.github/actions/action-a/
└ action.yml
```

- JavaScriptアクション
  - Node.jsで実行されるアクション

```
.github/actions/action-b/
├ index.js
├ package.json
└ action.yml
```

- Dockerコンテナーアクション
  - Dockerコンテナー内で実行されるアクション

```
.github/actions/action-c/
├ Dockerfile
├ entrypoint.sh
└ action.yml
```

カスタムアクションの種類と構成例

- 複合アクション（Composite actions）
  - 複数のステップをひとつにまとめたアクション

```
.github/actions/action-a/
└ action.yml
```

- JavaScriptアクション
  - Node.jsで実行されるアクション

```
.github/actions/action-b/
├ index.js
├ package.json
└ action.yml
```

- Dockerコンテナーアクション
  - Dockerコンテナー内で実行されるアクション

```
.github/actions/action-c/
├ Dockerfile
├ entrypoint.sh
└ action.yml
```

カスタムアクションのディレクトリにはactions.ymlが配置される。これは「アクションのメタデータファイル」と呼ばれ、名前や説明、入力・出力パラメータ、runs（カスタムアクションの種類の指定）などが含まれる

本コースでは複合アクションについて解説

カスタムアクションの種類と構成例

- 複合アクション（Composite actions）
  - 複数のステップをひとつにまとめたアクション

```
.github/actions/action-a/
└ action.yml
```

- JavaScriptアクション
  - Node.jsで実行されるアクション

```
.github/actions/action-b/
├ index.js
├ package.json
└ action.yml
```

- Dockerコンテナーアクション
  - Dockerコンテナー内で実行されるアクション

```
.github/actions/action-c/
├ Dockerfile
├ entrypoint.sh
└ action.yml
```

モジュール6

- カスタムアクションとは？
- カスタムアクションの公開
- カスタムアクションの種類と構成例
- 複合アクションの利用例
- まとめ

```yaml
name: ...
↵
on:↵
  ...↵
↵
jobs:↵
  build:↵
    runs-on: ubuntu-latest↵
    steps:↵
      - name: ...↵
        uses: actions/checkout@v4↵
      - name: ...↵
        uses: ...↵
      - name: ...↵
        uses: ...↵
      - name: ...↵
        run: |↵
          ...↵
          ...↵
      - name: ...↵
        run: |↵
          ...↵
          ...↵
↵
```

あるワークフローがある

```yaml
name: ...
↵
on:↵
  ...↵
↵
jobs:↵
  build:↵
    runs-on: ubuntu-latest↵
    steps:↵
      - name: ...↵
        uses: actions/checkout@v4↵
      - name: ...↵
        uses: ...↵
      - name: ...↵
        uses: ...↵
      - name: ...↵
        run: |↵
          ...↵
          ...↵
      - name: ...↵
        run: |↵
          ...↵
          ...↵
↵
```

この部分の複数ステップが
長過ぎるので
別ファイルに分割したい、とする

```yaml
name: ...
↵
on:↵
  ...↵
↵
jobs:↵
  build:↵
    runs-on: ubuntu-latest↵
    steps:↵
      - name: ...↵
        uses: actions/checkout@v4↵
      - name: ...↵
        uses: ...↵
      - name: ...↵
        uses: ...↵
      - name: ...↵
        run: |↵
          ...↵
          ...↵
      - name: ...↵
        run: |↵
          ...↵
          ...↵
↵
```

カスタムアクションのメタデータファイルを作成。

runs:
  using: "composite"
で、このアクションが複合アクションであることを示している

./github/actions/my-composite-action/action.yml

```yaml
name: ...↵
description: ...↵
↵
runs:↵
  using: "composite"↵
  steps:↵
    - name: ...↵
      uses: ...↵
    - name: ...↵
      uses: ...↵
    - name: ...↵
      run: |↵
        ...↵
        ...↵
    - name: ...↵
      run: |↵
        ...↵
        ...↵
↵
```

```yaml
name: ...
↵
on:↵
  ...↵
↵
jobs:↵
  build:↵
    runs-on: ubuntu-latest↵
    steps:↵
      - name: ...↵
        uses: actions/checkout@v4↵
      - name: ...↵
        uses: ...↵
      - name: ...↵
        uses: ...↵
      - name: ...↵
        run: |↵
          ...↵
          ...↵
      - name: ...↵
        run: |↵
          ...↵
          ...↵
↵
```

./github/actions/my-composite-action/action.yml

```yaml
name: ...↵
description: ...↵
↵
runs:↵
  using: "composite"↵
  steps:↵
    - name: ...↵
      uses: ...↵
    - name: ...↵
      uses: ...↵
    - name: ...↵
      run: |↵
        ...↵
        ...↵
    - name: ...↵
      run: |↵
        ...↵
        ...↵
↵
```

steps: の部分をカスタムアクション側へ移動

./github/actions/my-composite-action/action.yml

```yaml
name: ...
on:
  ...
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: ...
        uses: actions/checkout@v4
      - name: ...
        uses: ./.github/actions/my-composite-action
```

元の部分をカットし、代わりに複合アクションの呼び出しを書く。これでOK!

```yaml
name: ...
description: ...

runs:
  using: "composite"
  steps:
    - name: ...
      uses: ...
    - name: ...
      uses: ...
    - name: ...
      run: |
        ...
        ...
    - name: ...
      run: |
        ...
        ...
```

./github/actions/my-composite-action/action.yml

```yaml
name: ...
on:
  ...
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: ...
        uses: actions/checkout@v4
      - name: ...
        uses: ./.github/actions/my-composite-action
```

なおこのドット・スラッシュのドットは、「自分自身のリポジトリ」を表している。
ここを書き換えれば他のリポジトリも参照できる。

```yaml
name: ...
description: ...
runs:
  using: "composite"
  steps:
    - name: ...
      uses: ...
    - name: ...
      uses: ...
    - name: ...
      run: |
        ...
        ...
    - name: ...
      run: |
        ...
        ...
```

./github/actions/my-composite-action/action.yml

```yaml
name: ...
on:
  ...
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: ...
        uses: actions/checkout@v4
      - name: ...
        uses: ./.github/actions/my-composite-action
```

複合アクションを
利用する形になった。
記述が短くなり、見通しがよくなった

```yaml
name: ...
description: ...
runs:
  using: "composite"
  steps:
    - name: ...
      uses: ...
    - name: ...
      uses: ...
    - name: ...
      run: |
        ...
        ...
    - name: ...
      run: |
        ...
        ...
```

複合アクション
（ステップのあつまり）

./github/actions/my-composite-action/action.yml

```yaml
name: ...
on:
  ...
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: ...
        uses: actions/checkout@v4
      - name: ...
        uses: ./.github/actions/my-composite-action
```

```yaml
name: ...
description: ...
runs:
  using: "composite"
  steps:
    - name: ...
      uses: ...
    - name: ...
      uses: ...
    - name: ...
      run: |
        ...
        ...
    - name: ...
      run: |
        ...
        ...
```

複合アクション化の前後で、ステップの実行のされかたは同じ。ファイルが別になっただけ。これらのステップは、ランナーの中で順番に実行されていく。

モジュール6

- カスタムアクションとは？
- カスタムアクションの公開
- カスタムアクションの種類と構成例
- 複合アクションの利用例
- まとめ

まとめ

- **カスタムアクション**は、GitHubのユーザーが必要に応じて作成・利用できる、再利用可能な、処理の部品。
- GitHubリポジトリ内の .github/actions/(カスタムアクション名)/以下に配置される。
- **複合アクション**、**JavaScriptアクション**、**Dockerアクション**の3種類がある。
  - **複合アクション**では、ジョブ内の複数のステップを別ファイル（**ワークフローのメタデータファイル actions.yml 内**）に定義する。
  - **JavaScriptアクション**ではJavaScriptで処理を定義する。
  - **Dockerアクション**では、任意のコンテナー内で任意の言語を使用して処理を定義する。
- カスタムアクションはGitHub Marketplaceに公開して他の開発者に利用してもらうことも可能。

目次

- ラーニングパス1
  - モジュール1 GitHub Actions を使用して開発タスクを自動化する
  - モジュール2 GitHub Actions を使用して継続的インテグレーション ワークフローを構築する
  - モジュール3 GitHub Actions を使ったアプリケーションのビルドと Azure へのデプロイ
  - モジュール4 GitHub スクリプトを使用した GitHub の自動化
- ラーニングパス2
  - モジュール5 GitHub Actions を活用して GitHub Packages に公開する
  - モジュール6 カスタム GitHub アクションを作成して公開する
  - モジュール7 企業で GitHub Actions を管理する

モジュール7

- エンタープライズ（企業）向けのGitHub
- 再利用可能なワークフロー
- 再利用可能なワークフローの利用例
- 参考: 複合アクション vs 再利用可能ワークフロー
- 企業でのアクションの管理
- ランナーの種類・特徴・動作
- まとめ

モジュール7

- エンタープライズ（企業）向けのGitHub
- 再利用可能なワークフロー
- 再利用可能なワークフローの利用例
- 参考: 複合アクション vs 再利用可能ワークフロー
- 企業でのアクションの管理
- ランナーの種類・特徴・動作
- まとめ

エンタープライズ（企業）向けのGitHub

- GitHub Enterprise Cloud（GHEC）
  - クラウド型の企業向けGitHub
  - github.com 上で使用（インターネット接続が必要）
  - ユーザー管理には Enterprise Managed Users（EMU）を使用
- GitHub Enterprise Server（GHES）
  - オンプレミス型の企業向けGitHub
  - 自社でサーバーを準備してそこにGHSをセットアップして使用
  - インターネット接続が不要（閉域ネットワークで使用可能）
  - セキュリティポリシーが厳しい業界（金融・政府など）で使用される
- GitHub ActionsはGHECでもGHESでも利用可能
  - ただしGitHubホステッドランナーはGHESでは利用不可

モジュール7

- エンタープライズ（企業）向けのGitHub
- 再利用可能なワークフロー
- 再利用可能なワークフローの利用例
- 参考: 複合アクション vs 再利用可能ワークフロー
- 企業でのアクションの管理
- ランナーの種類・特徴・動作
- まとめ

再利用可能なワークフロー

- 企業内（組織内）で使用される共通のワークフロー（たとえばビルド・テスト・デプロイ）を「**再利用可能なワークフロー**」として定義し、共同利用することで、企業内の複数のリポジトリで同じようなワークフローを繰り返し書く必要がなくなる。

- メリット:
  - 企業内で処理を統一
  - ベストプラクティスを共有できる
  - メンテナンス性が向上

ワークフローを再利用する - GitHub ドキュメント

モジュール7

- エンタープライズ（企業）向けのGitHub
- 再利用可能なワークフロー
- 再利用可能なワークフローの利用例
- 参考: 複合アクション vs 再利用可能ワークフロー
- 企業でのアクションの管理
- ランナーの種類・特徴・動作
- まとめ

```yaml
name: Main Workflow

on:
  push:
    branches: [ "main" ]

jobs:
  use-reusable:
    uses: ./.github/workflows/reusable-build.yml   # ← 呼び出し
    with:
      node-version: '20'
    secrets:
      token: ${{ secrets.GITHUB_TOKEN }}
```

「再利用可能なワークフロー」を呼び出すワークフロー

再利用可能なワークフロー

```yaml
name: Reusable Build Workflow

on:
  workflow_call:   # ← 再利用可能にするためのトリガー
    inputs:
      node-version:
        required: true
        type: string
    secrets:
      token:
        required: true

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: ${{ inputs.node-version }}

      - name: Install dependencies
        run: npm install

      - name: Run tests
        run: npm test
```

```yaml
name: Main Workflow

on:
  push:
    branches: [ "main" ]

jobs:
  use-reusable:
    uses: ./.github/workflows/reusable-build.yml   # ← 呼び出し
    with:
      node-version: '20'
    secrets:
      token: ${{ secrets.GITHUB_TOKEN }}
```

「再利用可能なワークフロー」を呼び出すワークフロー側では、
uses: に、「再利用可能なワークフロー」のパスを書く。

この例では同じリポジトリ内のワークフローを参照しているが、組織内の別リポジトリにあるワークフローも参照できる

```yaml
name: Reusable Build Workflow

on:
  workflow_call:   # ← 再利用可能にするためのトリガー
    inputs:
      node-version:
        required: true
        type: string
    secrets:
      token:
        required: true

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: ${{ inputs.node-version }}

      - name: Install dependencies
        run: npm install

      - name: Run tests
        run: npm test
```

```yaml
name: Main Workflow

on:
  push:
    branches: [ "main" ]

jobs:
  use-reusable:
    uses: ./.github/workflows/reusable-build.yml   # ← 呼び出し
    with:
      node-version: '20'
    secrets:
      token: ${{ secrets.GITHUB_TOKEN }}
```

with: を使用して、「再利用可能なワークフロー」へパラメータを渡すことができる。

ここではNode.jsのバージョンを渡している。

```yaml
name: Reusable Build Workflow

on:
  workflow_call:   # ← 再利用可能にするためのトリガー
    inputs:
      node-version:
        required: true
        type: string
    secrets:
      token:
        required: true

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: ${{ inputs.node-version }}

      - name: Install dependencies
        run: npm install

      - name: Run tests
        run: npm test
```

```yaml
name: Main Workflow

on:
  push:
    branches: [ "main" ]

jobs:
  use-reusable:
    uses: ./.github/workflows/reusable-build.yml   # ← 呼び出し
    with:
      node-version: '20'
    secrets:
      token: ${{ secrets.GITHUB_TOKEN }}
```

secrets: を使用して、GITHUB_TOKEN（ワークフローの動作に必要な認証情報。GitHubが自動的に生成する）を渡している。

GITHUB_TOKENは、呼び出し先のワークローには、自動では引き継がれないため、このように明示的に渡す必要があることに注意。

```yaml
name: Reusable Build Workflow

on:
  workflow_call:   # ← 再利用可能にするためのトリガー
    inputs:
      node-version:
        required: true
        type: string
    secrets:
      token:
        required: true

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: ${{ inputs.node-version }}

      - name: Install dependencies
        run: npm install

      - name: Run tests
        run: npm test
```

```yaml
name: Main Workflow

on:
  push:
    branches: [ "main" ]

jobs:
  use-reusable:
    uses: ./.github/workflows/reusable-build.yml   # ← 呼び出し
    with:
      node-version: '20'
    secrets:
      token: ${{ secrets.GITHUB_TOKEN }}
```

「再利用可能なワークフロー」側では、on: に workflow_call: を書く。
またこのワークフローに渡す必要があるパラメータをinputs: で定義する。
さらにGITHUB_TOKENを受け取るため secrets: token: の記述も必要。

```yaml
name: Reusable Build Workflow

on:
  workflow_call:   # ← 再利用可能にするためのトリガー
    inputs:
      node-version:
        required: true
        type: string
    secrets:
      token:
        required: true

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: ${{ inputs.node-version }}

      - name: Install dependencies
        run: npm install

      - name: Run tests
        run: npm test
```

```yaml
name: Main Workflow

on:
  push:
    branches: [ "main" ]

jobs:
  use-reusable:
    uses: ./.github/workflows/reusable-build.yml   # ← 呼び出し
    with:
      node-version: '20'
    secrets:
      token: ${{ secrets.GITHUB_TOKEN }}
```

残りの部分は普通のワークフローと同じ。

```yaml
name: Reusable Build Workflow

on:
  workflow_call:   # ← 再利用可能にするためのトリガー
    inputs:
      node-version:
        required: true
        type: string
    secrets:
      token:
        required: true

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: ${{ inputs.node-version }}

      - name: Install dependencies
        run: npm install

      - name: Run tests
        run: npm test
```

モジュール7

- エンタープライズ（企業）向けのGitHub
- 再利用可能なワークフロー
- 再利用可能なワークフローの利用例
- 参考: 複合アクション vs 再利用可能ワークフロー
- 企業でのアクションの管理
- ランナーの種類・特徴・動作
- まとめ

参考: 複合アクション vs 再利用可能ワークフロー

- どちらも共通化・再利用の仕組み

- 複合アクション: 簡易的な共通化
  - ジョブの中の複数ステップを括り出してアクション化するしくみ
  - 複合アクション内のステップは呼び出し元と同じランナーで実行される
  - secret（GITHUB_TOKEN）を渡す必要はない

- 再利用可能ワークフロー: 大規模な共通化
  - ワークフロー（＝一つまたは複数のジョブ）全体を再利用するしくみ
  - 再利用可能ワークフロー内の各ジョブは、それぞれ別のランナーで実行される
  - secret（GITHUB_TOKEN）を渡す必要がある

モジュール7

- エンタープライズ（企業）向けのGitHub
- 再利用可能なワークフロー
- 再利用可能なワークフローの利用例
- 参考: 複合アクション vs 再利用可能ワークフロー
- 企業でのアクションの管理
- ランナーの種類・特徴・動作
- まとめ

企業でのアクションの管理

- 企業向けGitHub（GHEC・GHES）では、**ポリシー**を使用して、**組織で利用できるアクションと再利用可能ワークフロー**を設定できる

モジュール7

- エンタープライズ（企業）向けのGitHub
- 再利用可能なワークフロー
- 再利用可能なワークフローの利用例
- 参考: 複合アクション vs 再利用可能ワークフロー
- 企業でのアクションの管理
- ランナーの種類・特徴・動作
- まとめ

ランナーの種類

- GitHub ホステッド ランナー
  - GitHubによって管理されるランナー
  - Linuxランナー、Windowsランナー、macOSランナーが選べる

- セルフホステッドランナー
  - ユーザーが所有、メンテナンスするランナー（サーバー）
  - 必要に応じてサーバーのスペックや環境をカスタマイズできる

セルフホステッド ランナー - GitHub ドキュメント

GitHub ホステッド ランナー - GitHub ドキュメント

ランナーの特徴

- GitHub ホステッド ランナー
  - ◯ユーザーによるメンテナンスは不要
  - ◯基本的には有料サービスだが、条件付きで無料でも使える

- セルフホステッドランナー
  - ◯GitHubホステッドランナーには導入されていないソフトウェアを導入できる
  - △セットアップ、セキュリティ対策、監視が必要
  - △GitHubとしての料金はかからないが、サーバーの導入コスト・運用コストはユーザー負担となる

セルフホステッド ランナー - GitHub ドキュメント

GitHub ホステッド ランナー - GitHub ドキュメント

ランナーの動作

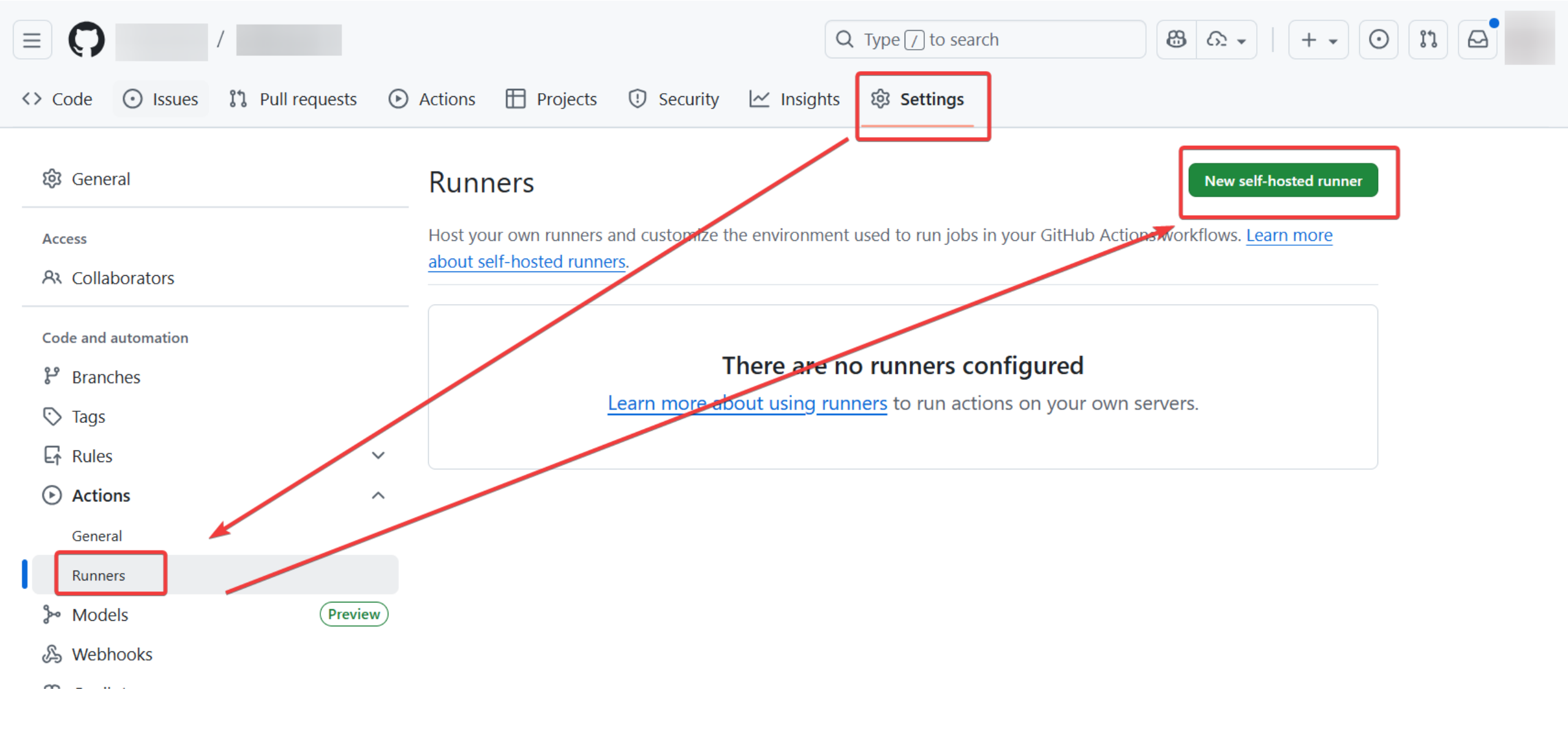
- GitHub ホステッド ランナー
  - ジョブの実行は毎回新しいランナーで行われる
  - ジョブの実行が終わるとそのランナーは破棄される

- セルフホステッドランナー
  - ジョブの実行が終わっても特にランナーの破棄は行われない

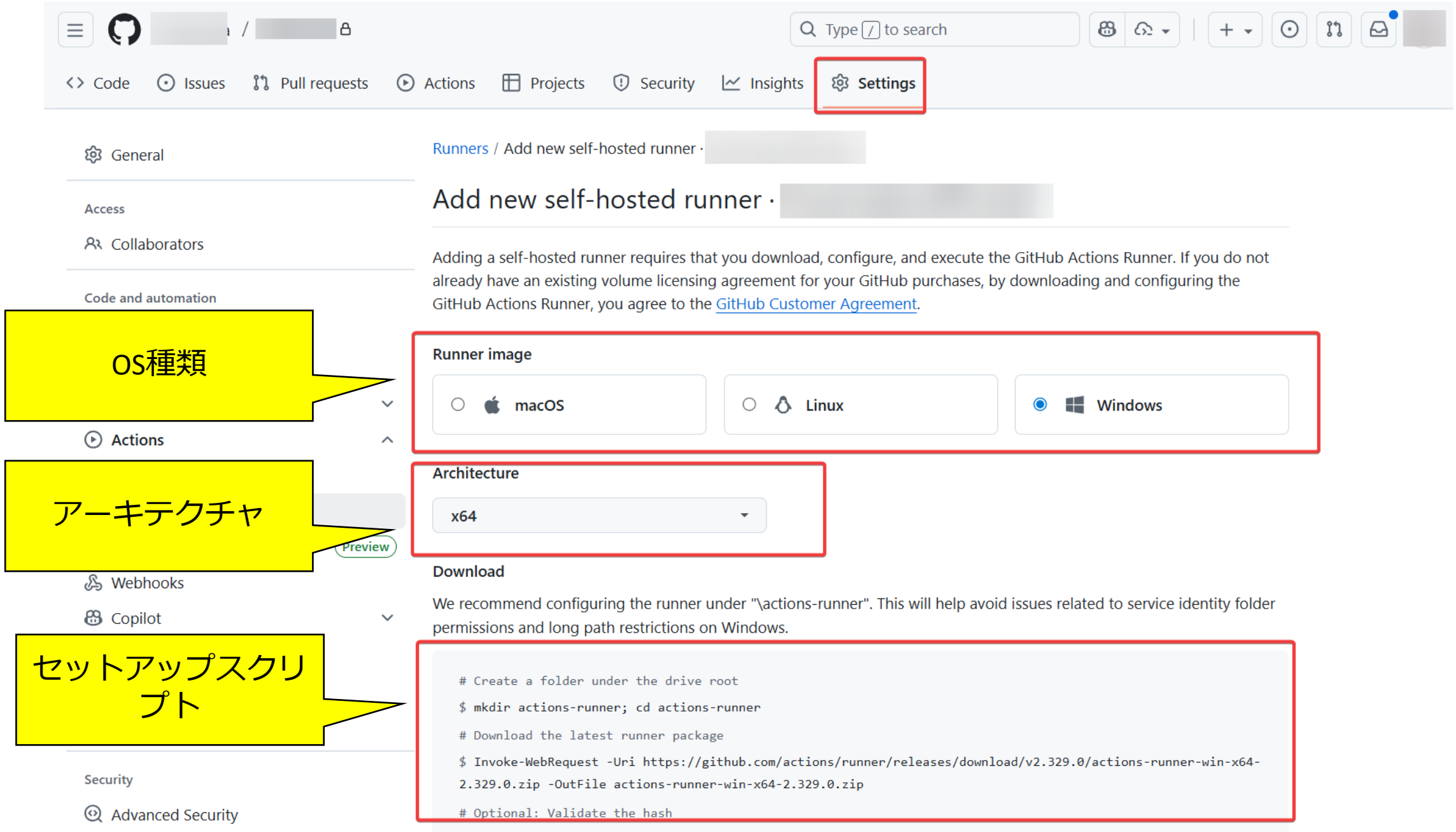
セルフホステッド ランナー - GitHub ドキュメント

GitHub ホステッド ランナー - GitHub ドキュメント

リポジトリで、セルフホステッドランナーを追加する例
※企業での利用の場合、GitHub組織の「Settings」からランナーを追加し組織で共同利用も可



OS種類、アーキテクチャを選択し、セットアップスクリプトを入手。
セルフホステッドランナーとするサーバー上でこのスクリプトを実行する



OS種類

アーキテクチャ

セットアップスクリプト

モジュール7

- エンタープライズ（企業）向けのGitHub
- 再利用可能なワークフロー
- 再利用可能なワークフローの利用例
- 参考: 複合アクション vs 再利用可能ワークフロー
- 企業でのアクションの管理
- ランナーの種類・特徴・動作
- まとめ

まとめ

- 企業向けのGitHubとして「**GitHub Enterprise Cloud**」（GHEC）と「**GitHub Enterprise Server**」（GHES）が利用できる。GHECはクラウド型で、すばやく導入できる。GHESはオンプレミス型で、各企業でサーバーを準備して導入する。GHESは閉域ネットワークでの利用が可能。
- 「**再利用可能なワークフロー**」を使用することで、組織内で、ワークフローを一元管理・共有できる。
- **ポリシー**を使用して、組織で利用できるアクションと再利用可能ワークフローを設定（制限）できる。
- ランナーの種類として「**GitHubホステッドランナー**」と「**セルフホステッドランナー**」がある。「GitHubホステッドランナー」はGHESでは利用できない。「セルフホステッドランナー」は運用管理の手間がかかるが、無料で利用でき、自由なセットアップが可能。

全体のまとめ

- モジュール1 では、GitHub Actions のワークフロー、イベント、ジョブ、ステップ、アクションについて解説しました
- モジュール2 では、継続的インテグレーションを実装する方法を解説しました
- モジュール3 では、Azure App ServiceへWebアプリをデプロイするアクションについて解説しました
- モジュール4では、github-scriptを使用してGitHub の操作を自動化する方法を解説しました
- モジュール5 では、GitHub PackagesとGitHub Container Registryについて解説しました
- モジュール6 では、カスタムアクションを作成する方法を解説しました
- モジュール7 では、企業でのGitHub Actions利用、再利用可能なワークフロー、ポリシー、ランナーの種類について解説しました