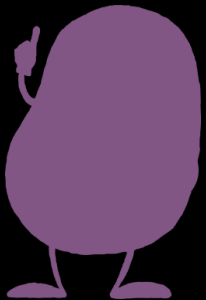


AWSとGitHubを使ってみよう勉強会

～ 第3回 GitHub Actionsを使ったCI環境の構築 ～

株式会社 豆蔵
ビジネスソリューション事業部



本日の内容

- 前回の課題の回答(10分)
- 課題の補足(50分)
 - 解説 GitHub Actions
 - 解説 GitHub Packages
- 次回までの課題の説明(20分)

画面キャプチャやコマンド操作等はGitHubのssi-mz-studygroupユーザで行った例となります。キャプチャやコマンド等の該当部分は自分のユーザIDに読み替えてください

- ssi:simple_server infra
- mz:mamezou

前回の課題の回答



前回の課題：(再掲)

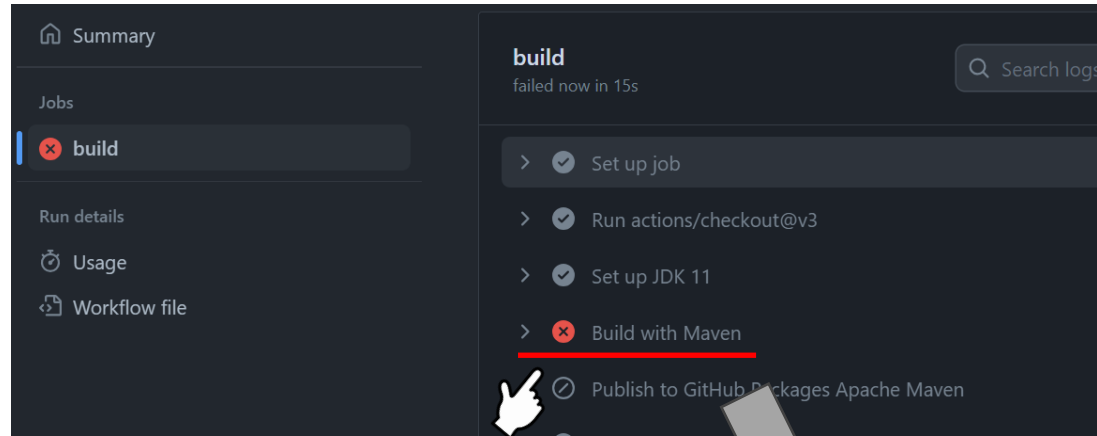
- 課題その 1

- ビルドエラーにならないようにワークフロー定義を修正する
- 後続スライドのpom.xmlへの追加設定を行う
- ヒント
 - サンプルのアプリで使っているHelidonはJava17を必要とします
 - uses: actions/setup-java@v3のようにusesプロパティの右側にあるものはJP1の組み込みJOBのようなものに相当します。なので、何者か？やどんなプロパティ設定があるのか？などは“actions/setup-java”で検索すると分かると思います

- 課題その 2

- リポジトリに変更があった場合、つまりmainブランチになにかがコミットされたら自動でワークフローが実行されるようにする
- ヒント
 - on:プロパティに起動条件を追加
 - これもグーグル検索すれば沢山情報があると思います

課題その1：エラーの確認



ビルドでエラー

展開してログを追っていくと

```
904 [INFO] Compiling 3 source files to /home/runner/work/sample-app/sample-app/target/classes
905 [INFO] -----
906 [INFO] BUILD FAILURE
907 [INFO] -----
908 [INFO] Total time: 8.006 s
909 [INFO] Finished at: 2023-07-23T05:46:17Z
910 [INFO] -----
911 Error: Failed to execute goal org.apache.maven.plugins:maven-compiler-plugin:3.1:compile
hello-app: Fatal error compiling: error: invalid target release: 17 -> [Help 1]
```

release17(Java17)が必要なのに対して
コンパイラターゲットが不正

課題その1 : workflow定義の確認と修正

<修正前>

```
steps:
- uses: actions/checkout@v3
- name: Set up JDK 11
  uses: actions/setup-java@v3
  with:
    java-version: '11'
    distribution: 'temurin'
    server-id: github
    settings-path: ${github.workspace}
```

ヒントにあったようにサンプルのビルドにはJava17が必要だが、
workflowでビルドするためにセットアップしたのはJava11のため


<修正後>

```
steps:
- uses: actions/checkout@v3
- name: Set up JDK 17
  uses: actions/setup-java@v3
  with:
    java-version: '17'
    distribution: 'temurin'
    server-id: github
    settings-path: ${github.workspace}}
```

17を指定して実行することでエラーは解消

課題その2： mainブランチになにかがコミットされたら自動でワークフローを実行する

```
on:
  release:
    types: [created]
  workflow_dispatch:
  push:
    branches:
      - 'main'
```



回答の追加部分

参考：色々な設定

```
push:
  branches:
    - 'main'
    - 'mona/octocat'
    - 'releases/**'
  tags:
    - v2
    - v1.*
```

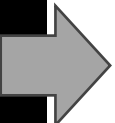
- onはworkflowの起動条件
- この起動条件はGitHub Actionsに予めいくつも条件が用意されている
- 今回はこの中のpushを使う
 - pushはリポジトリになんらかpushされたらworkflowを起動する
 - branchesを指定しない場合はすべてのブランチが対象となる
 - 今回はmainブランチが対象なのでbranchesに明示的に監視対象を指定する
- pushには↓のように色々指定することができる（or条件）

- main という名前のブランチ
- mona/octocat という名前のブランチ
- releases/10 のように名前が releases/ で始まるブランチ
- v2 という名前のタグ
- v1.9.1 のように名前が v1. で始まるタグ

課題その2：おまけ

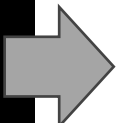
• 便利な起動条件

```
on:
  pull_request:
    branches:
      - main
```



- main ブランチに対するプルリクエストが発行されたら起動する
- プルリクエストはGitHub独自の機能で、意味的には「マージリクエスト」といった方が理解しやすい
- gitflowやgithubflowなどGitを使ったブランチ戦略では変更チケット単位にブランチを作成し、開発ブランチなどの幹となるブランチに細かくマージしていく
- マージはブランチから直接行うのではなく「このブランチの修正内容を幹のブランチにマージをお願いします」というプルリクエストを発行し、一般的には管理者的な人が修正内容を確認してマージを行う
- よって、プルリクエストはマージの承認行為に相当するため、発行されたタイミングでテストの実行や静的解析の実行などを行うのが定石で、この起動条件を使って実現する

```
on:
  schedule:
    - cron: '30 5 * * 1,3'
```



- 見たままのcron
- ただし、空きリソースがない場合は実行開始がdelayされる
- 経験則的に普通に1分～30分のレベルで遅れる

GitHub Actionsはマニュアルが非常に充実していて、かつ分かりやすい。なので、何かないかな？どうやるのかな？といった場合、まず[マニュアル](#)を見てみるのがよい

（この反対がAWS。AWS文学と言われるほど、マニュアルを読むと分らないことが余計分からなくなる・・・）

課題の補足

- ・解説 GitHub Actions
- ・解説 GitHub Packages



GitHub Actionsとは

- GitHub Actionsとは
 - JP1やJCLなどと同じようにジョブの実行を制御するGitHub Actions独自の仕組み
 - JP1ではJOB定義で、JCLではJCLで「どのように実行するか」を定義するが、GitHub Actionsも同様に独自の構文で定義する
 - この独自の構文で「どのように実行するか」をYAMLで定義したものをGitHub Actionsではworkflowと呼ぶ

<GitHub Actionsのworkflowの例>

```
name: Maven Package
on:
  release:
    types: [created]
  workflow_dispatch:
  push:
jobs:
  build:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      packages: write
    steps:
      - uses: actions/checkout@v3
```

```
- name: Set up JDK 11
  uses: actions/setup-java@v3
  with:
    java-version: '11'
    distribution: 'temurin'
    server-id: github
    settings-path: ${github.workspace}
- name: Build with Maven
  run: mvn -B package --file pom.xml
- name: Publish to GitHub Packages Apache Maven
  run: mvn deploy -s $GITHUB_WORKSPACE/settings.xml
  env:
    GITHUB_TOKEN: ${github.token}}
```



GitHub Actionsの構造 - 1/2

`name: Maven Package` workflow名

`on:`
`release:`
`types: [created]` 起動条件
`workflow_dispatch:`

`jobs:` Job全体

`build:`
`runs-on: ubuntu-latest`
`permissions:` Job名がbuildの
`contents: read` Jobの定義
`packages: write`

workflow

`steps:` Stepの定義

`- uses: actions/checkout@v3`

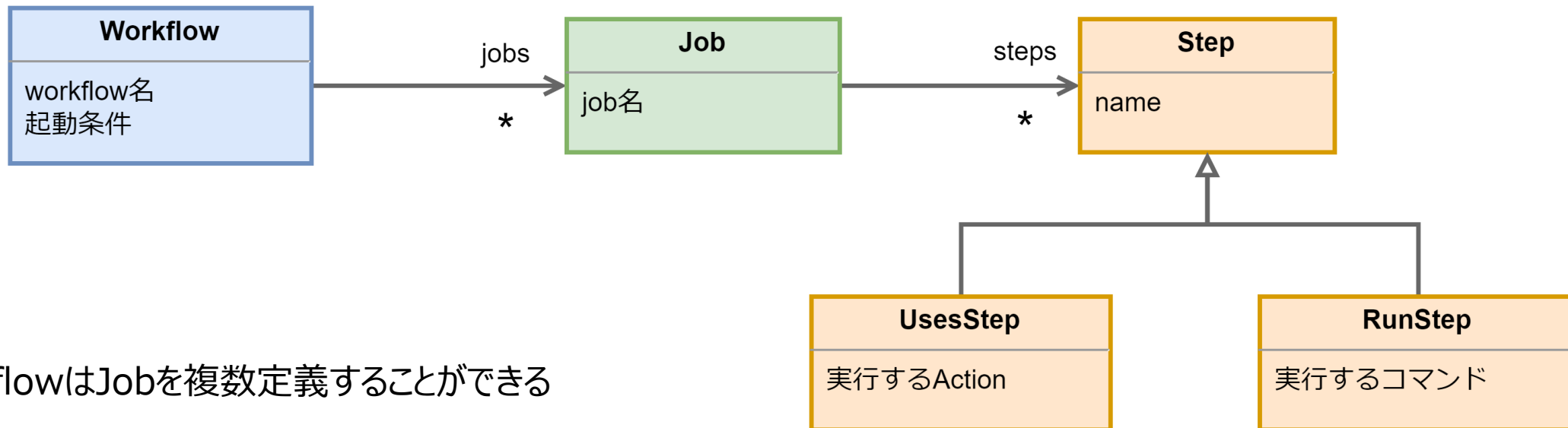
`- name: Set up JDK 11`
`uses: actions/setup-java@v3`
`with:`
`java-version: '11'`
`distribution: 'temurin'`
`server-id: github`
`settings-path: ${{ github.workspace }}`

`- name: Build with Maven`
`run: mvn -B package --file pom.xml`

`- name: Publish to GitHub Packages Apache Maven`
`run: mvn deploy -s $GITHUB_WORKSPACE/settings.xml`
`env:`
`GITHUB_TOKEN: ${{ github.token }}`

個々のStepの定義

GitHub Actionsの構造 – 2/2



- workflowはJobを複数定義することができる
- Jobは1つの実行単位
 - 厳密にはJobごとに新しい仮想環境が割り当てられる (Cleanビルドの実現)
 - 個々のJobは独立しているため、あるJobの処理結果を別のJobから参照するといったことはできない (引き継ぐ仕組みを使えば可能)
 - Jobの実行が完了するとその仮想環境は破棄される (mavenのlocalリポジトリも含め何も残らない)
- Jobは複数のStep(処理)によって構成される
 - Stepで実行できるのはGitHub Actions独自の仕組みでモジュール化されたActionかシェルコマンドのどちらか
 - 双方とも1Stepで実行できるのは1アクションまたは1コマンドのみ (コマンドは1ライナーならいくらでも可)

課題のworkflowの内容 - 1/2

```
name: Maven Package
```

```
on:
```

```
  release:
```

```
    types: [created]
```

```
  workflow_dispatch:
```

```
jobs:
```

```
  build:
```

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

```
    permissions:
```

```
      contents: read
```

```
      packages: write
```

- Jobを実行するOSの指定
- ubuntu, Windows, Macの指定が可能
- だが、課金料金は ubuntu < Windows < Macとなる（Freeプランではubuntuは33.3時間/月まで無料でMacはubuntuの10倍換算）
- ubuntu-20.04などと特定のバージョンを指定することも可能（ただしGitHubに用意されているもののみ）
- セルフホスランナーの仕組みを使いオンプレに作成した環境を使うことも可能（ただし難易度高）

実行JOBに割り当てる権限

- contents : issueなども含めOwnerのGitHubコンテンツに対する権限
- packages : jarをデプロイするGitHub Packageに対する権限

※今回はpackagesへのデプロイがあるため明示的に指定したが、書き込みする操作がなければ明示的に指定する必要なし

課題のworkflowの内容 - 2/2

steps:

- uses: actions/checkout@v3

- name: Set up JDK 17
uses: actions/setup-java@v3

with:

java-version: '17'
distribution: 'temurin'
server-id: github
settings-path: \${github.workspace }

- name: Build with Maven
run: mvn -B package --file pom.xml

- name: Publish to GitHub Packages Apache Maven
run: mvn deploy -s \$GITHUB_WORKSPACE/settings.xml
env:
GITHUB_TOKEN: \${github.token }

- checkoutアクションを使ってリポジトリからコードをチェックアウトする
(どのリポジトリをどこになどのアクションの詳細は後述)

- setup-javaアクションを使ってJava17をセットアップする
- Jobに割り当てられたコンテナ環境にJava temurin Runtime 17をインストールして環境変数やMavenの設定準備をする (パラメータの詳細は後述)

- runで指定されたコマンドを実行する
- この例はmavenで最終的に/target配下にjarファイルを生成する

- runで指定されたコマンドを実行する
- この例はmavenで前stepで生成された/target配下のjarファイルをPackageレジストリにデプロイする

<actionとは>

- actionはworkflowで行う処理を再利用するためのGitHub Actions独自の仕組み。
- 実体はJavaScriptまたはShellScriptできている。GitHubやAWSなどの大きな組織から個人まで様々なものが公開されている

actionのマニュアルを見てみよう

- actionは高機能なことをやってくれるがイマイチなにをやっているのか分からない場合や細かい調整がしたい場合はマニュアルを見てみるとよい
- “actions/checkout@v3”でググって最初に出てくる開発元の[GitHubリポジトリ](#)にしてみる
 - なにをやってくれるかは概要から、そしてデフォルトで何をどこにチェックアウトしてくるかはUsageをみると分かる
 - repositoryが未指定なので取ってくるリポジトリは`${{ github.repository }}`で、これもググるとでてくるがworkflowが動作しているリポジトリとなる
 - チェックアウトする場所を指定するpathは未定なので、`$GITHUB_WORKSPACE`となり、これもググるとでてくるが、デフォルトでは作業ディレクトリの`/home/runner/work/`となる
- “actions/setup-java@v3”も興味があればググって調べてみると理解が深まると思います
- このようにGitHub Actionsは再利用可能な手順をモジュール化したactionを組み合わせてJobを定義できるとことが魅力です
- つい最近ではAWSのCodeBuildがGitHub Actionsのactionをサポートするようになり話題となりました
 - AWSにはactionのような手順の再利用手段がないためこれを実現したかった模様

GitHub Actionsで可能な Job制御

- 一番シンプルな例を説明してきましたが、JP1までとはいきませんが、かなり色々な制御ができます（CI/CD用途では困ることはないレベル）
- これについては、以下の記事が分かりやすいのでそちらを使って説明します
 - [GitHub Actions ワークフローにおけるジョブ制御 | 豆蔵デベロッパーサイト](#)
- デモ
 - needsを使って先行Jobと後続をJobを分けた例
 - `try-aws-github-learning/.github/workflows/demo-1_jobsplit-with-needs.yml`
 - MatrixStrategyという素敵な機能を使った例（スゴいですよー）
 - `try-aws-github-learning/.github/workflows/demo-2_job-matrix.yml`

課題の補足

- ・解説 GitHub Actions
- ・解説 GitHub Packages



GitHub Packagesとは

- GitHubが提供するPackageレジストリサービス
- JavaのJarやDockerのコンテナイメージなどPackageとして登録できるものは複数ある
 - GitHub Package はサービスの総称で登録するものにより実体も使い方も微妙に異なる
 - Jarを登録するのはGitHub Packages Apache Maven レジストリ
 - コンテナイメージを登録するのはGitHub Packages コンテナレジストリ
 - 前回の課題で使っていたのはGitHub Packages Apache Maven レジストリで、今回の課題ではGitHub Packages コンテナレジストリを使う
- GitHub Packages Apache Maven レジストリの役割や機能はインハウスリポジトリで使っているnexnusと同じ（ただし機能は少ない）
- GitHub Packages コンテナレジストリの役割や機能はコンテナレジストリとしてよく使われるDocker Hubと同じ
- 一般的にPackageのデプロイには認証などの複雑な仕組みや設定が必要となるが、GitHub Actionsからは簡易に使える⇒実際にどう便利かは次のスライドで
- GitHub Packages Apache Maven レジストリからのPackageの取得にはpublicなものであってもGitHubの認証が必要だが、GitHub Packages コンテナレジストリはpublicなイメージであれば認証は不要
 - なので、GitHub Packages コンテナレジストリは最近よく使われている

今回の課題の設定を試みる

<workflow定義>

```
- name: Set up JDK 17
  uses: actions/setup-java@v3
  with:
    java-version: '17'
    distribution: 'temurin'
    server-id: github
    settings-path: ${github.workspace}

- name: Build with Maven
  run: mvn -B package --file pom.xml

- name: Publish to GitHub Packages Apache Maven
  run: mvn deploy -s $GITHUB_WORKSPACE/settings.xml
```

- ② mavenの実行時にはactionで生成された settings.xmlを参照（これがないとdeployできない）

① 設定をもとにactionでオンザフライで作業ディレクトリ直下に生成される

<生成されるMavenのsettings.xml>

```
<settings ...
  <servers>
    <server>
      <id>github</id>
      <username>${env.GITHUB_ACTOR}</username>
      <password>${env.GITHUB_TOKEN}</password>
    </server>
  </servers>
</settings>
```

GitHub Packagesへのアクセストークンが自動で設定される

インハウスリポジトリへアクセスするため、みなさんのJDEVにもsettings.xmlの設定はあるはず

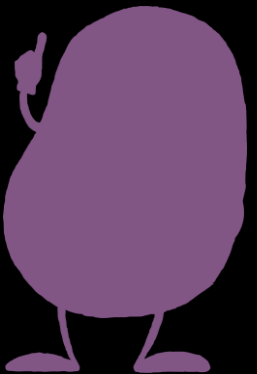
③ 登録先の解決

```
<distributionManagement>
  <repository>
    <id>github</id>
    <name>GitHub Packages</name>
    <url>https://maven.pkg.github.com/ssi-mz-studygroup/sample-app</url>
  </repository>
</distributionManagement>
```

<追加してもらったpomの定義>



次回までの課題の説明



次回までの課題

- テーマ
 - 今回までの課題の成果をもとにサンプルアプリをコンテナ化する
 - GitHub Actionsを使ってJavaのビルドからコンテナイメージの作成、レジストリへのデプロイを行う（CIからCDまでやってみる）
 - ここで作成したコンテナイメージを次々回ではAWSで動作させます
- お題
 - アプリをコンテナ化するDockerfileを作る
 - Javaのビルドからコンテナイメージの作成、レジストリへのデプロイまで行うworkflowをひな形からコピーして動作させる
- ゴール
 - 作成したコンテナイメージが正しく動作すること
 - ワークフローの実行が成功する
 - GitHub Packagesにコンテナイメージがデプロイ(登録)されていること

課題の実施手順

今回の課題はCodespacesとWeb UIの双方を使います

Step1. ヒントをもとにDockerfileを作成する

Step2. コンテナイメージをビルドして動作を確認する

Step3. ワークフローをひな形からコピーする

Step4. docker-pluginの設定をpomに追加する

Step5. ワークフローを実行する

Step1. ヒントをもとにDockerfileを作成する

- リポジトリ直下にファイル名を"Dockerfile"にして以下のヒントをもとにDockerfileを完成させる

```
# ベースイメージはeclipse-temurin(旧OpenJDK)のJava17を使用
FROM docker.io/eclipse-temurin:17-jre-alpine

# 作業ディレクトリを/(root)にする
...TODO
# Mavenのビルド成果物(hello-app.jar)をコンテナイメージにCOPY
...TODO
# Mavenのビルド成果物(libs以下を)をコンテナイメージにCOPY
...TODO

# ExecutableJarをjavaコマンドで起動
...TODO
```



Step2. コンテナイメージをビルドして動作を確認する

Codespaces

- [ひな形プロジェクト](#)のREADMEの「Dockerfileの確認（Dockerfileを作ってから行う）」に従い動作を確認する

```
# jarのビルド
mvn clean package
# コンテナイメージのビルド
docker build -t hello-app .
# ローカルリポジトリに登録されていることの確認(hello-appがあること)
docker images hello-app
# コンテナの実行
docker run -p 7001:7001 --name hello-app --rm hello-app
# REST APIへのリクエスト(別のターミナルから)
curl -X GET localhost:7001/api/hello
```


Step3. ワークフローをひな形からコピーする

Codespaces

- mamezou-tech/try-aws-github-learning/.github/workflows/image-publish.ymlと同じものを自分のリポジトリに作成する
- Step1で作ったDockerfileとimage-publish.ymlをリポジトリにcommitしてpushする

Step4. docker-pluginの設定をpomに追加する

- mamezou-tech/try-aws-github-learning/pom.xmlの設定をもとに自分のpomにdocker-pluginの設定を追加します

```
<project xmlns=http://maven.apache.org/POM/4.0.0...>
...
<properties>
...
<!-- docker plugin settings -->
<image.tag>latest</image.tag>
</properties>
...
<build>
<finalName>${project.artifactId}</finalName>
<plugins>
...
<!-- dockerプラグインの設定 -->
<plugin>
<groupId>io.fabric8</groupId>
<artifactId>docker-maven-plugin</artifactId>
<version>0.40.2</version>
<configuration>
<registry>ghcr.io</registry>
<images>
<image>
↓は自分のgithubユーザ名に変更する
<name>ssi-mz-studygroup/hello-app</name>
<build>
<tags>
<tag>${image.tag}</tag>
</tags>
<contextDir>${project.basedir}</contextDir>
<cleanup>try</cleanup>
</build>
</image>
</images>
</configuration>
</plugin>
</plugins>
</build>
...
</project>
```

自分のpomにそのまま追加

自分のpomに追加。
その際、赤色下線分のssi-mz-studygroupの部分は
[自分のgithubユーザ名]/hello-appとなるように変更する

設定の追加が完了したら修正したpomをリポジトリにcommitとして
pushする



Step5. ワークフローを実行する

Web UI

- Actionsからimage-publishを実行して成功を確認する



参考情報

- 今さら聞けないMaven – コンテナも一緒にビルドしたい。テスト実行前にコンテナを起動したい | 豆蔵デベロッパーサイト
 - https://developer.mamezou-tech.com/blogs/2022/08/31/docker_with_maven/
 - 「テスト実行前にコンテナを起動したい」の章は関係なし
- GitHub Packages Container Registryをモデリングしてみた – UMLを理解の道具として | 豆蔵デベロッパーサイト
 - <https://developer.mamezou-tech.com/blogs/2023/03/09/ghcr-modeling/>

次回の予定



次回の予定

- 今回の課題の説明
 - 補足説明と質疑応答
- 次回までの課題の説明
 - テーマはGitHub ActionsでデプロイしたコンテナイメージをAWSで動かす
(ついにAWS登場！)