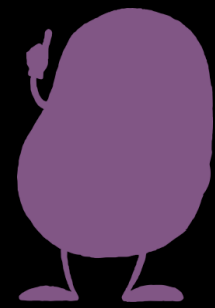
AWSとGitHubを使ってみよう勉強会
~ 第2回 GitHub CodespacesとHelidonの利用 ~

株式会社 豆蔵
ビジネスソリューション事業部

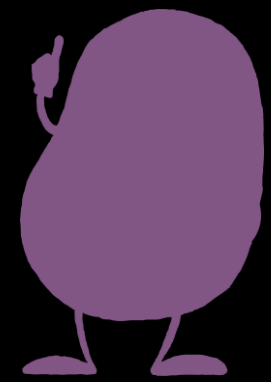


updated:2023/9/10

本日の内容

- 前回の課題の回答(5分)
- 課題の補足(35分)
  - GitHub Codespacesとは
  - Sampleで使っているHelidonの仕組み
  - Sampleのテストの仕組み
  - Gitの補足
- 次回までの課題の説明(20分)

前回の課題の回答



前回の課題
- (再掲)Step4. コードを修正してテストをパスさせる -

MAMEZOU-TECH

前回の課題
- テストコード側を見てみる -

<テストコード>

```java
@HelidonTest
@AddConfig(key = "server.port", value = "7001")
@ExtendWith(JulToSLF4DelegateExtension.class)
public class HelloResourceTest {

    private HelloResource helllResource;

    @BeforeEach
    public void setup() throws Exception {…

    @Test
    void tesHello() {
        var expected = "Hello";
        var actual = helllResource.hello();
        assertEquals(expected, actual);
    }
```

期待値

呼び出し

<プロダクトコード>

```java
@GET
@Produces(MediaType.TEXT_PLAIN)
public String hello() {
    return null; // TODO テストが通るように実装する
}
```

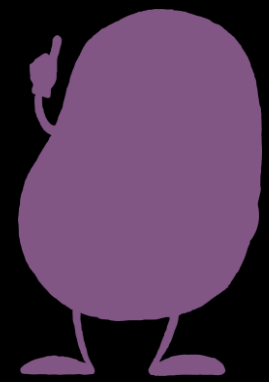
正解は "Hello"

```java
@GET
@Produces(MediaType.TEXT_PLAIN)
public String hello() {
    return "Hello";
}
```

MAMEZOU-TECH

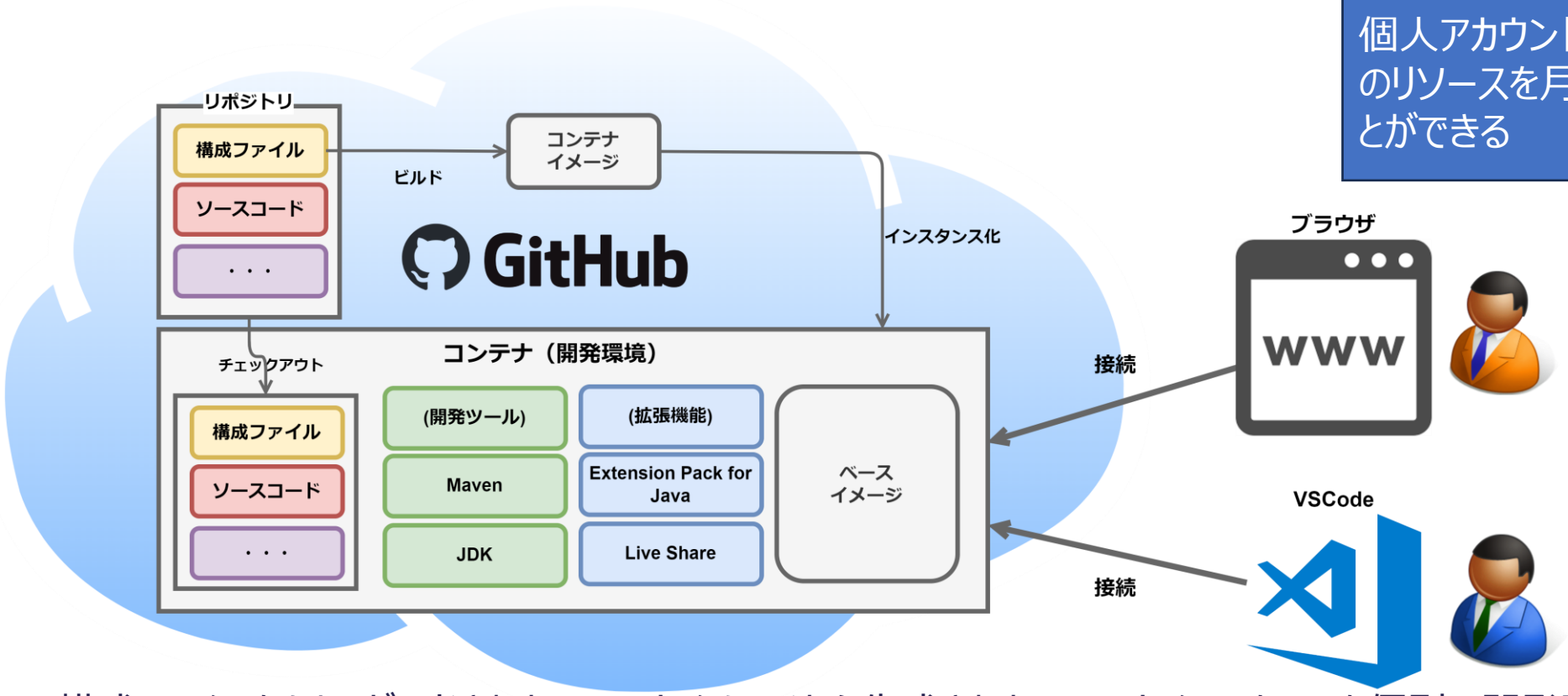
課題の補足

- GitHub Codespacesとは
- Sampleで使っているHelidonの仕組み
- Sampleのテストの仕組み
- Gitの補足



GitHub Codespacesとは

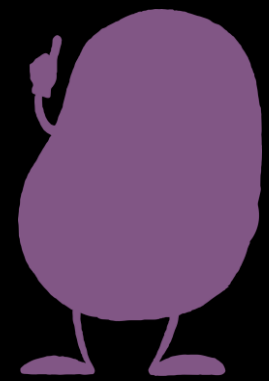
個人アカウントでも2コアCPU/4GBメモリのリソースを月60時間まで無料で使うことができる



- 構成ファイルをもとにビルドされたコンテナイメージから生成されたコンテナインスタンスを個別の開発環境としてユーザに提供する機能
- ユーザはブラウザもしくはローカルのVSCodeから生成されたコンテナインスタンスに接続して作業を行う
- ユーザは目の前にあるブラウザやVSCodeを操作するが、リポジトリからチェックアウトしたファイルやJavaのビルドや実行などはすべてGitHub側のコンテナ内に存在し行われます

課題の補足

- GitHub Codespacesとは
- Sampleで使っているHelidonの仕組み
- Sampleのテストの仕組み
- Gitの補足



Sampleを動かしてみる(1/2)

- Sampleで使っているHelidonとは
  - アプリケーションサーバ不要でJakartaEE(Core Profile)と**MicroProfile**アプリケーションを起動させることができるフレームワーク

デモしてみる

1. アプリケーションのビルド

```
@ssi-mz-studygroup ➜/workspaces/sample-app (theme/0711-2nd) $ mvn clean package -DskipTests
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------< sample:hello-app >------------------------
[INFO] Building hello-app 0.0.1-SNAPSHOT
[INFO]   from pom.xml
[INFO] --------------------------------[ jar ]---------------------------------
```

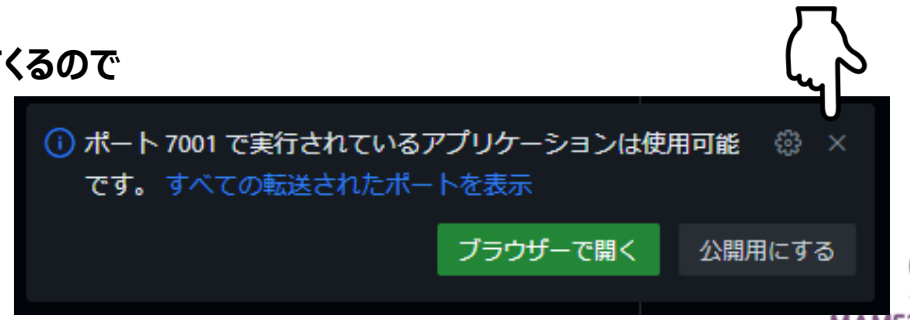
ここで紹介するコマンドは
https://github.com/ssi-mz-studygroup/sample-app
のREADMEにも記載しています

2. アプリケーションの実行

```
@ssi-mz-studygroup ➜/workspaces/sample-app (theme/0711-2nd) $ java -jar target/hello-app.jar
2023-07-08 06:08:05,134 [INFO ] [i.h.c.LogConfig] [main] - Logging at initialization configured using defau
2023-07-08 06:08:05,186 [INFO ] [o.j.w.Version] [main] - WELD-000900: 4.0.2 (Final)
2023-07-08 06:08:05,731 [INFO ] [o.j.w.Bootstrap] [main] - WELD-ENV-000020: Using jandex for bean discovery
2023-07-08 06:08:05,969 [INFO ] [o.j.w.Bootstrap] [main] - WELD-000101: Transactional services not availabl
2023-07-08 06:08:06,081 [INFO ] [o.j.w.Event] [main] - WELD-000411: Observer method [BackedAnnotatedMethod]
```

案内が出てくるので

とりあえず
無視して閉じる



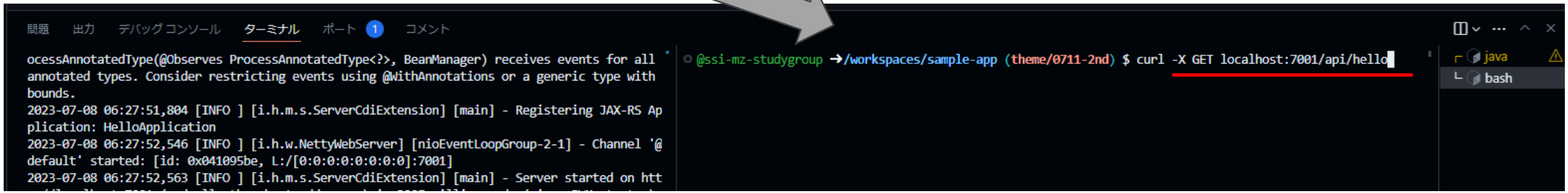
MAMEZOU-TECH

Sampleを動かしてみる(2/2)

3. コンソールの分割

エンドポイントがなぜ
GET localhost:7001/api/hello
になるかは分かりますよね？

4. リクエスト送信



MAMEZOU-TECH

Sampleが動作している環境

GitHub

コンテナインスタンス（Codepsace）内

ブラウザ

接続

Point

起動したJavaプロセスはGitHub側のコンテナインスタンス内で起動している。
よって、localhostは自分のローカルマシンのlocalhostではなく、コンテナインスタンスのlocalhostとなる。

コンソールで
$ ps aux
を試してみると見てくるものがあるかも！？

MicroProfileとは – Helidonが実装しているもの

＜MicroProfileプロジェクトの公式説明＞

MicroProfile®プロジェクトは、マイクロサービスアーキテクチャ向けにエンタープライズJavaを最適化することを目的とし、JakartaEEかを問わずJavaのエコシステムを活用しながらマイクロサービス向けの新しい共通APIと機能をオープンな環境で短いサイクルで提供してくことを目標にする

出典:https://projects.eclipse.org/projects/technology.microprofile

＜超訳＞

JakartaEEや既存のエコシステムを活用したマイクロサービスアーキテクチャ(MSA)向けのAPIや機能をオープンなプロセスで策定し、短期間でリリースしてく

つまるところ・・・

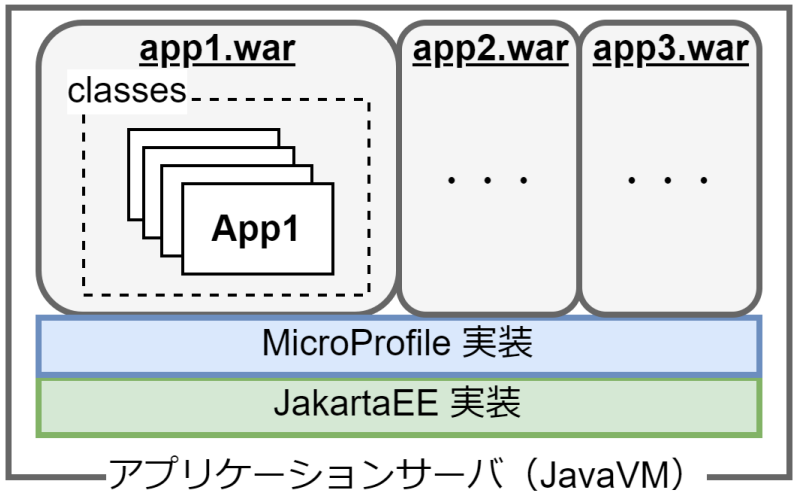
JakartaEEをベースとしたMSA向けAPIセット

MAMEZOU-TECH

MicroProfileの実装形態

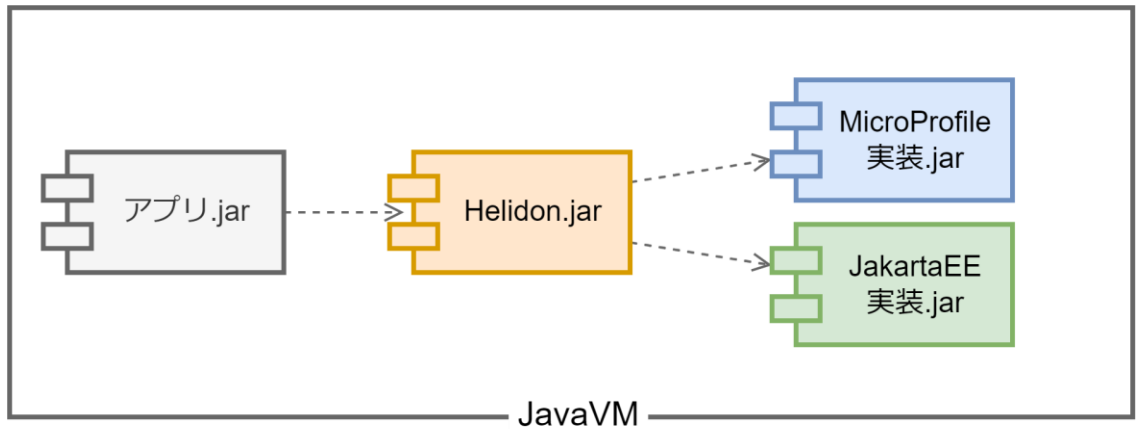
■ ２つの実装形態がある

＜JakartaEEアプリケーションサーバ(Liberty)＞



- ✓ アプリケーションサーバに内包されるMicroProfileの実装を利用する
- ✓ JakartaEEアプリと同じようにwar形式でアプリケーションサーバにデプロイする
- ✓ アプリの動作にはアプリケーションサーバが必要

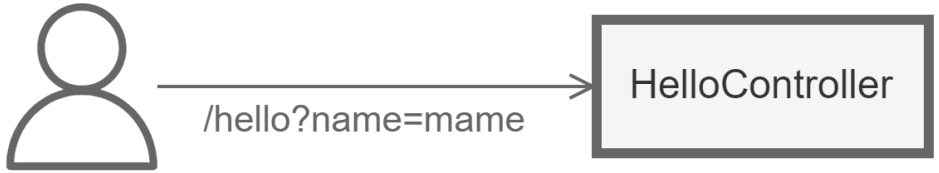
＜MicroProfileフレームワーク(Helidon)＞



- ✓ jarのライブラリ形式で提供されるMicroProfile実装をアプリケーションのdependencyに追加して利用する
- ✓ nettyなどの組み込みのHTTPサーバを内包しているため、アプリケーションサーバは不要で動作する
- ✓ 利用イメージとしては組込みTomcatを使ったSpringBootと同じ

HelidonはMicroProfileフレームワークの1つ

おまけ：Helidonの実装方法



＜アプリのコード＞

```java
@ApplicationScoped
@Path("/hello")
public class HelloController {
    @GET
    public String hello(@QueryParam("name") String name) {
      return String.format("Hello %s!", name);
    }
}
```

```java
package jaxrs;

public class HelloMain {
    public static void main(String[] args) {
        // Helidonの起動
        io.helidon.microprofile.cdi.Main.main(args);
    }
}
```

mainメソッドから起動

＜pom.xml(一部抜粋)＞

```xml
<dependencies>
  <dependency>
    <groupId>io.helidon.microprofile.bundles</groupId>
    <artifactId>helidon-microprofile-core</artifactId>
  </dependency>
  ...
</dependencies>
```

これだけで推移的依存により必要な依存が追加される

```xml
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-jar-plugin</artifactId>
  <configuration>
    <archive>
      <manifest>
        <addClasspath>true</addClasspath>
        <classpathPrefix>libs</classpathPrefix>
        <mainClass>jaxrs.HelloMain</mainClass>
      </manifest>
    </archive>
  </configuration>
</plugin>
```

mainClassとclasspathをMANIFESTファイルに設定した
ExecutableJarの作成

おまけ：Helidonのビルド～実行

1. mavenでアプリをビルド

```
$ mvn clean package
```

ビルド結果

```
target
  ├─jjug-mp-sample.jar <-- mainClassを持つアプリのExcutableJar
  ├─libs               <-- 依存ライブラリ(Helidonの実体)
  │    ├─aopalliance-repackaged-3.0.3.jar
  │    ├─helidon-common-3.2.1.jar
  │    ├─helidon-common-configurable-3.2.1.jar
  │    ├─helidon-common-context-3.2.1.jar
  │    ├─helidon-common-crypto-3.2.1.jar
  │    ├─helidon-common-http-3.2.1.jar
  │    ...
```

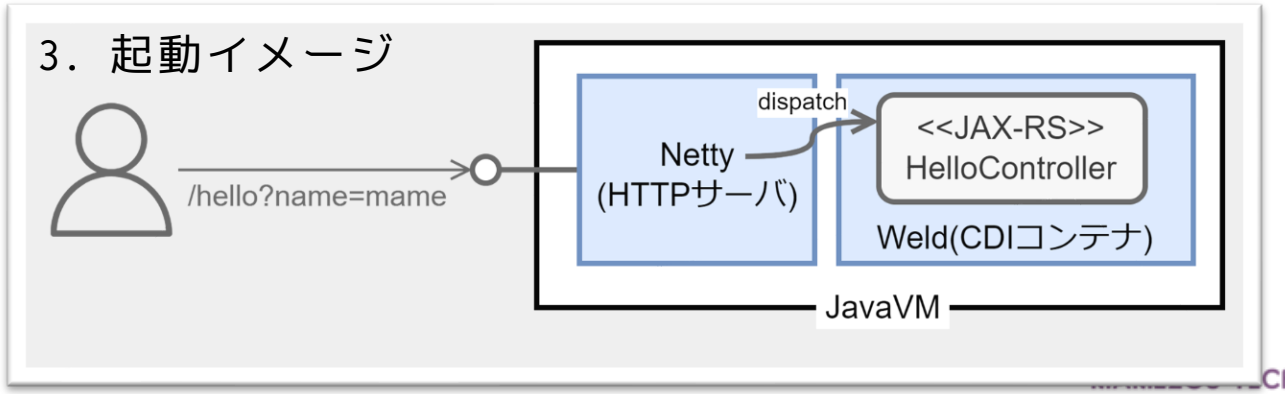
maven-dependency-pluginの
copy-dependenciesゴールで
runtimeとcompileスコープのjarが
./target/libにコピーされる

※target/libをls するとあります

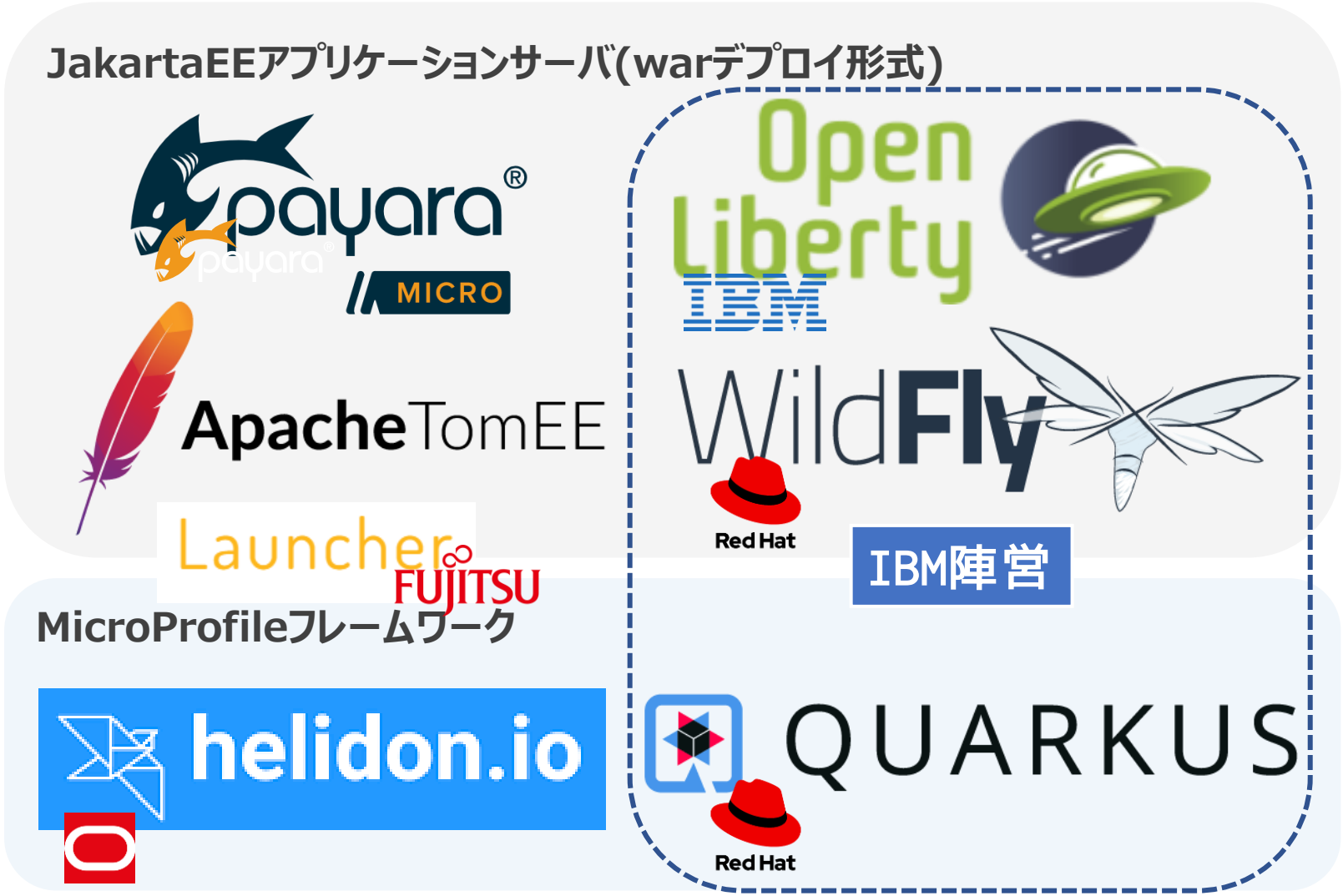
2. javaコマンドでアプリを起動

```
$ java -jar target/jjug-mp-sample.jar
```

3. 起動イメージ



主なMicroProfile実装

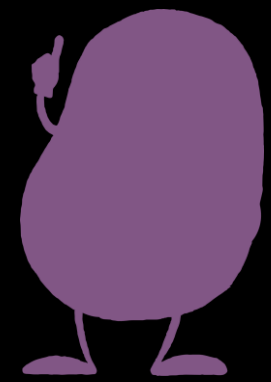


- ✓ 最新のMicroProfile6.0に対応しているのは現時点でOpenLibertyのみ
- ✓ ここに挙げたOpenLiberty以外はMicroProfile5.0に対応
- ✓ OpenLibertyは常にMicroProfileの最新仕様に追従するスタンス
- ✓ 図からもIBM陣営はMicroProfileに積極的なことがうかがえる

- ✓ LauncherはJakartaEEアプリケーションサーバではないが、アプリケーションをwar形式にする必要がある。warをuber JARに変換してJavaコマンドから起動することができる
- ✓ PayaraMicroも同様なことができる

課題の補足

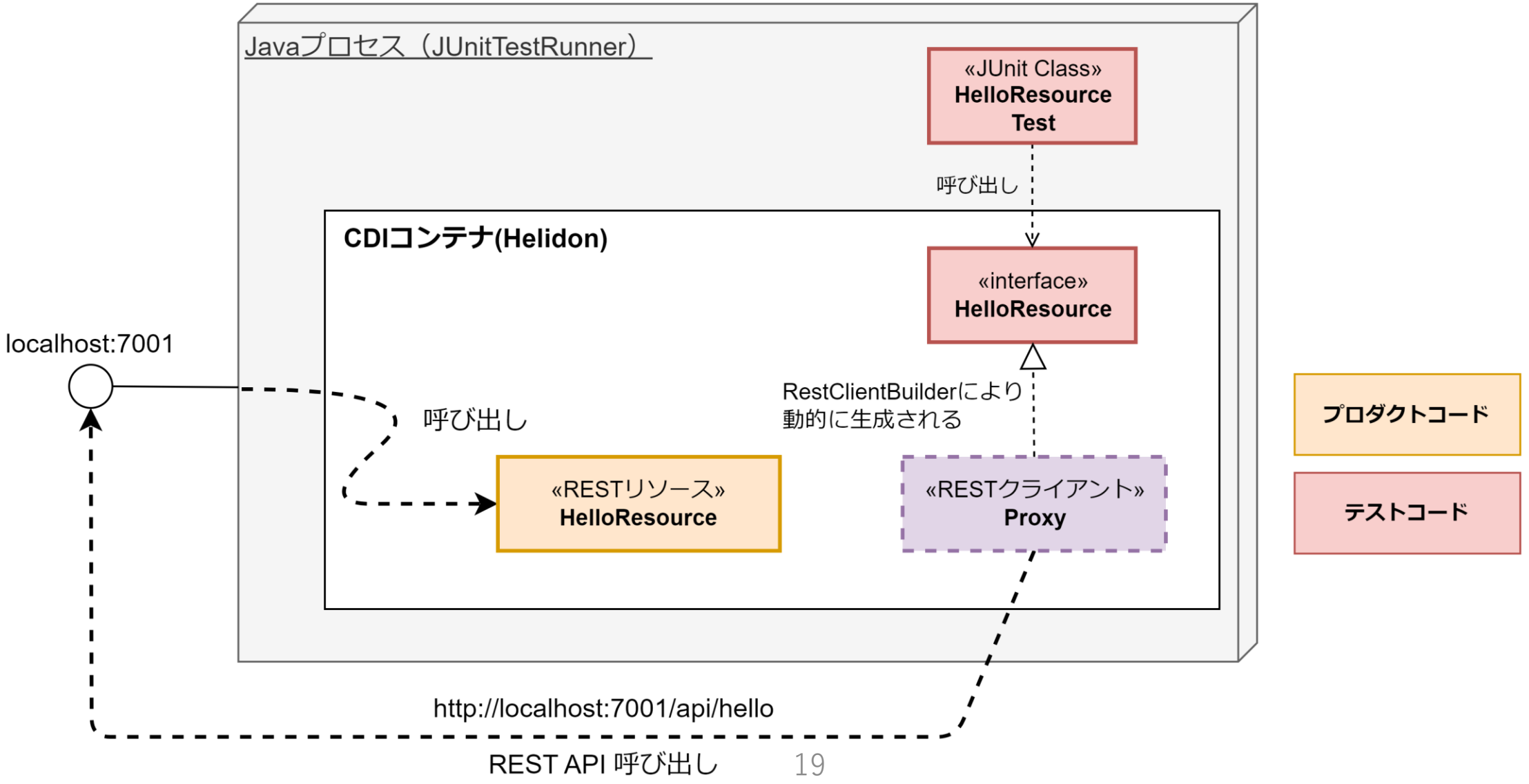
- GitHub Codespacesとは
- Sampleで使っているHelidonの仕組み
- Sampleのテストの仕組み
- Gitの補足



JAX-RSの単体テストについて

- WAS Libertyなどのアプリケーションサーバを使った開発ではJAX-RSアプリはアプリケーションサーバ上でしか動作させることができない。このため、JUnitを使って簡単にテストすることができず、この点がかなりつらい(Spring Bootが裏山)
- 一方Helidonはmainメソッドから普通にJAX-RSアプリを起動することができる。このため、HelidonのようなMicroProfileフレームワークでは単体テストを簡単に行うことができる (Spring Bootと方式は同等！)
- 今回のテストはJUnitから本物のJAX-RSアプリを動作させてテストを行っている

単体テストの仕組み – プロセスイメージ



MAMEZOU-TECH

単体テストの仕組み – テストコード

```java
@HelidonTest
@AddConfig(key = "server.port", value = "7001")
@ExtendWith(JulToSLF4DelegateExtension.class)
public class HelloResourceTest {

    private HelloResource helllResource;

    @BeforeEach
    public void setup() throws Exception {
        helllResource = RestClientBuilder
                .newBuilder()
                .baseUri(new URI("http://localhost:7001/api"))
                .build(HelloResource.class);
    }

    @Test
    void tesHello() {
        var expected = "Hello";
        var actual = helllResource.hello();
        assertEquals(expected, actual);
    }

    @Path("hello")
    interface HelloResource {
        @GET
        @Produces(MediaType.TEXT_PLAIN)
        String hello();
    }
}
```

Helidonを起動するアノテーション(HelidonのJUnit拡張)
Helidonを7001ポートで起動する指定
Helidonのログを見やすくするJUnit拡張（荻原オリジナル）

setupメソッドで取得したRESTクライアント。このRESTクライアントを使って、テスト対象のREST API(HelloResource)を呼び出す

MicroProfile RestClientの機能を使って、テスト対象のREST API(HelloResource)を呼び出すHelloResourceインタフェースに対するProxyインスタンスを取得する

HelloResourceインタフェースのProxyインスタンスから http://localhost:7001/api/helloのリクエストが発行され、本物のHelleoResource#helloが呼び出される

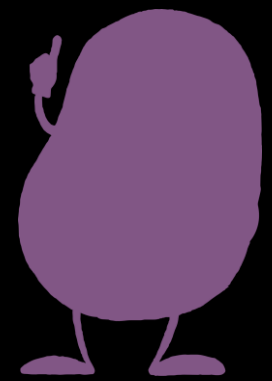
同様にREST APIを呼び出すかをJAX-RSのアノテーションを使って定義したMicroProfile RestClientのインタフェース定義
これはプロダクトコード側のsample.HelloResourceは別もの（インタフェース名を同じにしているが異なっていてもよい）

参考情報

- らくらくMicroProfile RestClient | 豆蔵デベロッパーサイト
  - https://developer.mamezou-tech.com/msa/mp/cntrn07-mp-restclient/
- Helidon Tips - MicroProfile RestClientを使ったRESTリソースのJUnitテスト | 豆蔵デベロッパーサイト
  - https://developer.mamezou-tech.com/msa/mp/ext03-helidon-rest-testing/

課題の補足

- GitHub Codespacesとは
- Sampleで使っているHelidonの仕組み
- Sampleのテストの仕組み
- Gitの補足



GitとGitHub

- GitはVCS(Version Control System)の1つ
- 未来で使っているVCSはSubversion
- GitもSubversionと同様に自分でインストールしてGit単独で使うこともできる

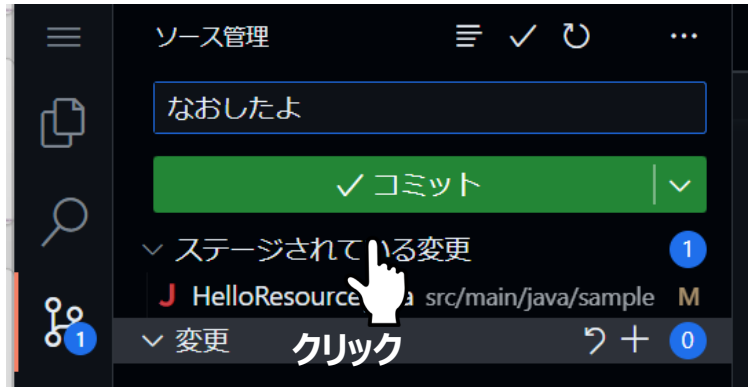
- GitHubはVCSとしてGitや課題管理、CI環境などソフトウェア開発に必要な様々なサービスを一体として提供してくれるSaaSアプリケーション
- 同様なSaaSアプリケーションとしてメジャーなものにGitLabがある

GitとSubversionの違い

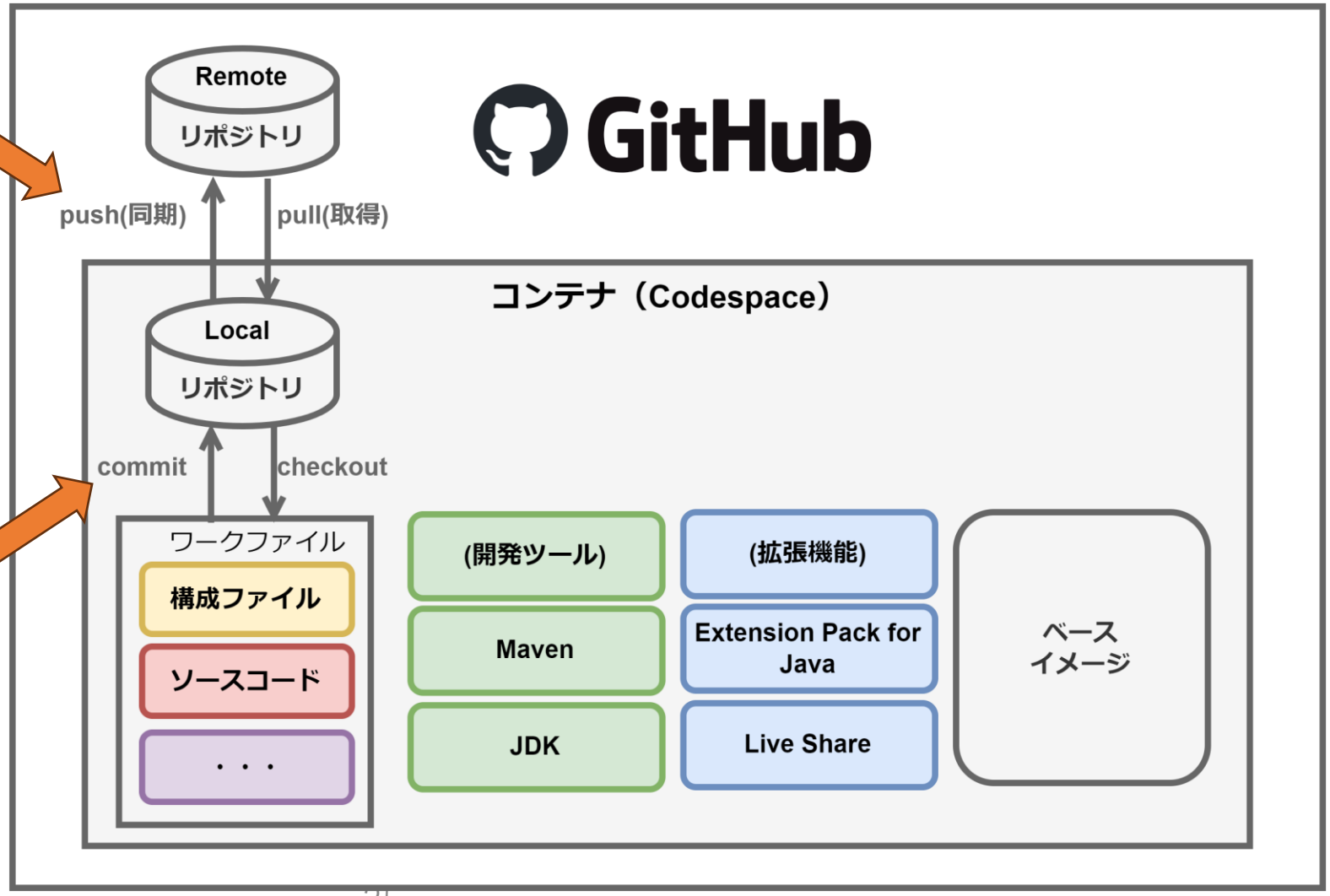
- Gitはマスタに相当するRemoteリポジトリとそのRemoteリポジトリをローカルに複製(clone)したLocalリポジトリの2種類のリポジトリから構成される
  - 作業者はLocalリポジトリからチェックアウトしたワークファイルに対して変更を行い、行った変更をLocalリポジトリにcommitする
  - Localリポジトリのcommit(変更内容)はRemoteリポジトリに対してpushすることでRemoteとLocalが同期させる
- これに対してSubversionはRemoteリポジトリのみ
  - Remoteリポジトリからチェックアウトしたワークファイルを変更し、Remoteリポジトリに直接コミットする
- Gitの作業はすべてリポジトリ単位
  - Subversionはリポジトリの下のどこからでもチェックアウトやブランチ、タグの作成ができるが、Gitではこのようなことはできない。
  - Gitでのチェックアウトやブランチ、タグの作成を行う単位はすべてリポジトリ単位となる

CodespacesとGitの操作の関係

Remoteリポジトリへのpush



Localリポジトリへのコミット



GitとSubversionの比較

Subversionの方が優れている点

- Subversionは仕組みがシンプル
  - Subversionは集中型でリポジトリがひとつです。**Gitは**リモートとローカルリポジトリの両方を意識しなければならず、**Subversionよりも操作が複雑です**。
- リビジョン番号がわかりやすい
  - commitするとcommitを識別するためのリビジョン番号（ID）が振られます。Subversionは連番で分かりやすいですが、Gitでは分散リポジトリのためIDにハッシュ値が使われ分かり辛いです。

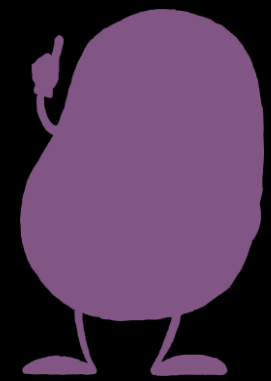
Gitの方が優れている点

- Gitはローカルにcommitできる
  - Gitは分散型でローカルにリポジトリを持つため、共有リポジトリを汚さずにローカル上でcommit出来ます。変更内容を共有リポジトリに反映させるかは任意です。Subversionは「実験的なコードや未完成なコードで共有リポジトリを汚したくない」との声がよく聞かれます
- ブランチとマージが使いやすい
  - **Gitのブランチとマージは Subversionに比べ高速かつ手軽に使えます**。Gitを活用するポイントはこの2つを使いこなす事です。Git flowやGitHub Flow等の開発フローが参考になります。

引用：https://tracpath.com/works/development/subversion_vs_git/

MAMEZOU-TECH

次回までの課題の説明



次回までの課題

- テーマ
  - GitHubのCI/CD機能であるGitHub Actionsを試してみる
  - GitHub Actions でコンパイル～デプロイまでの行ってみる
  - この成果物(artifact)をもとに次々回ではアプリのコンテナ化を行う
- お題
  - GitHubから提供されているGitHub Actions Template(Publish Java Package with Maven)からワークフロー定義を作成する
  - テンプレートを修正してゴールを満たすようにする
- ゴール
  - ワークフロー定義が課題の中で指定された条件を満たしている
  - ワークフローの実行が成功する
  - GitHub Packagesにjarがデプロイ(登録)されていること

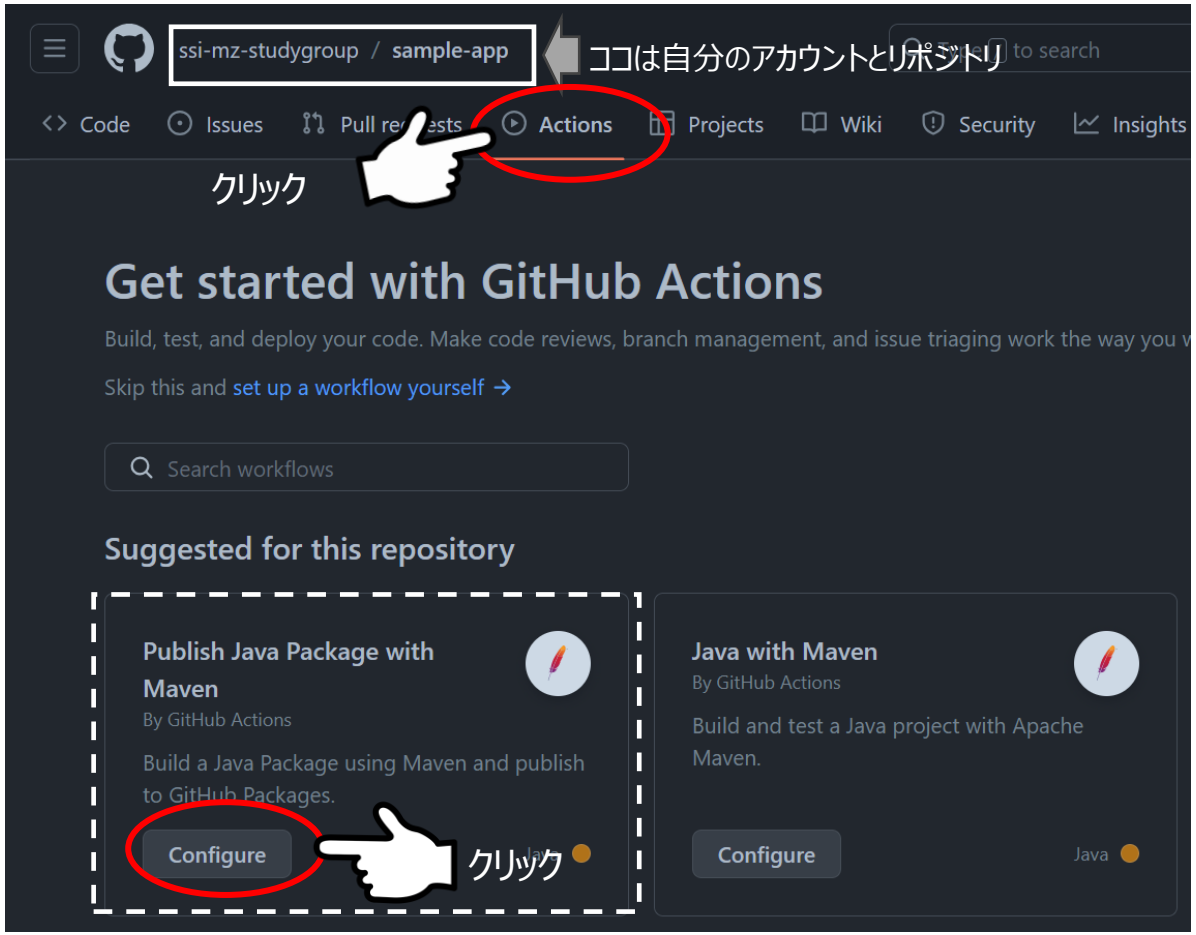
課題の実施手順

今回の課題はCodespacesを使いません

Step1. テンプレートからベースとなるワークフロー定義を生成する
Step2. 起動条件に手動実行を追加してコミットする
Step3. 実行してみる（失敗する）
Step4. 課題を対応する
Step5. 実行してみる（成功するまで行う）

Step1. テンプレートからベースとなるワークフロー定義を生成する

- リポジトリは前回の課題で各自に作成してもらったものを引き続き使います。⇒自分のリポジトリに移動

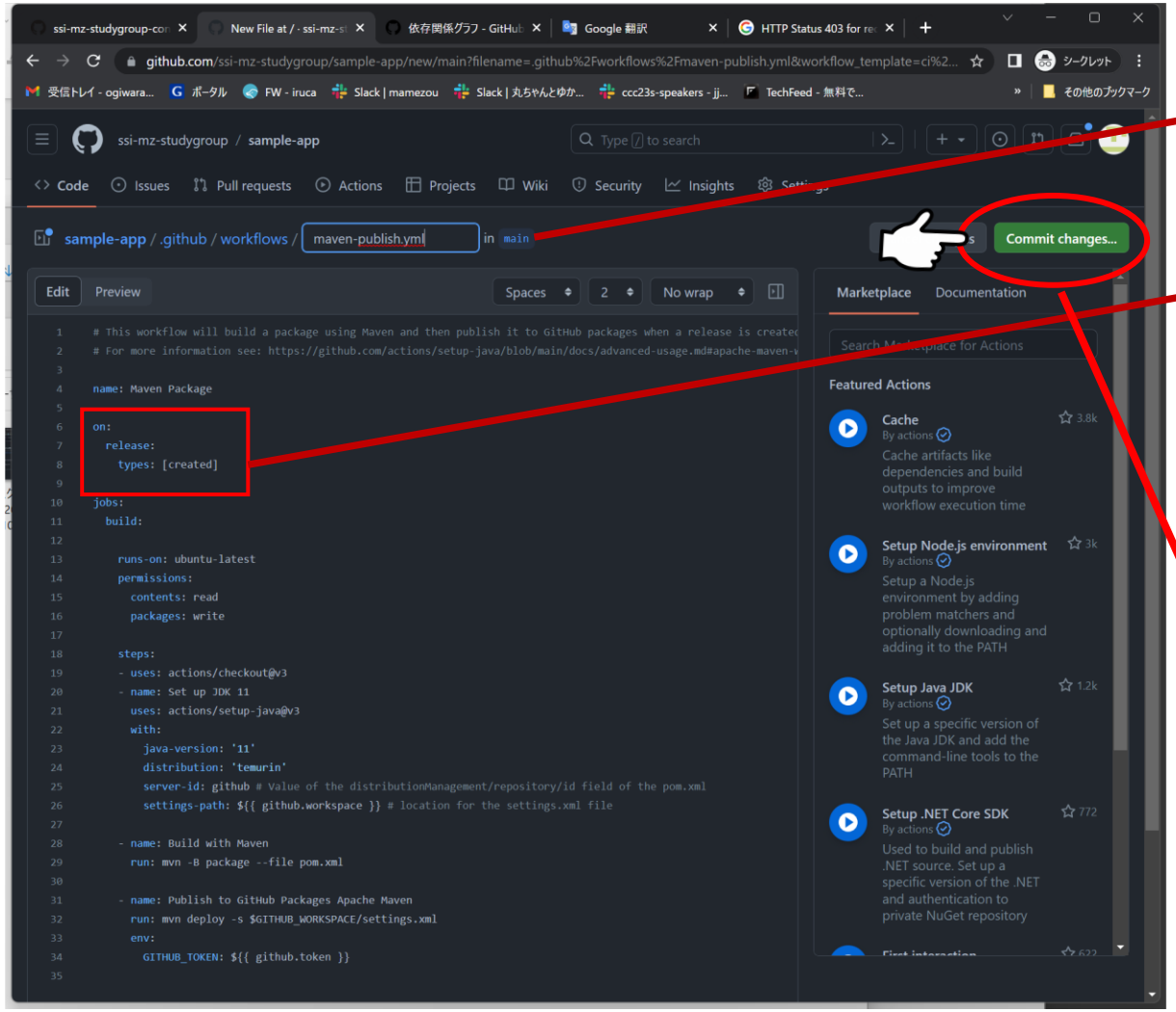


テンプレート選択画面から
・リポジトリのチェックアウト
・コンパイル＆jarの作成
・GitHub Packagesへのデプロイ
のアクションが予め定義された Publish Java Package with Mavenテンプレートを選択する

MAMEZOU-TECH

Step2. 起動条件に手動実行を追加してコミットする

- テンプレートに従ったワークフロー定義が生成される



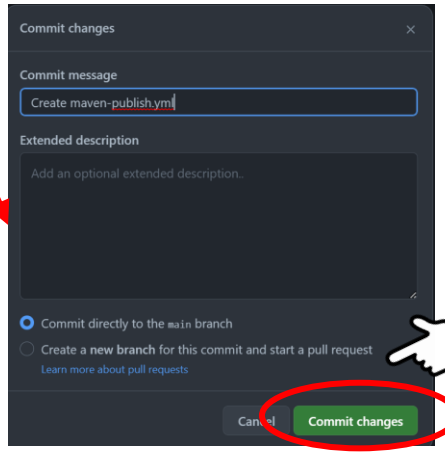
拡大

ワークフロー定義のファイル名の指定⇒そのままでOK

拡大

```
on:
  release:
    types: [created]
  workflow_dispatch:
```

起動条件(on:)に破線赤枠の内容(手動実行)を追加する
※YAMLフォーマットなのでインデントには注意



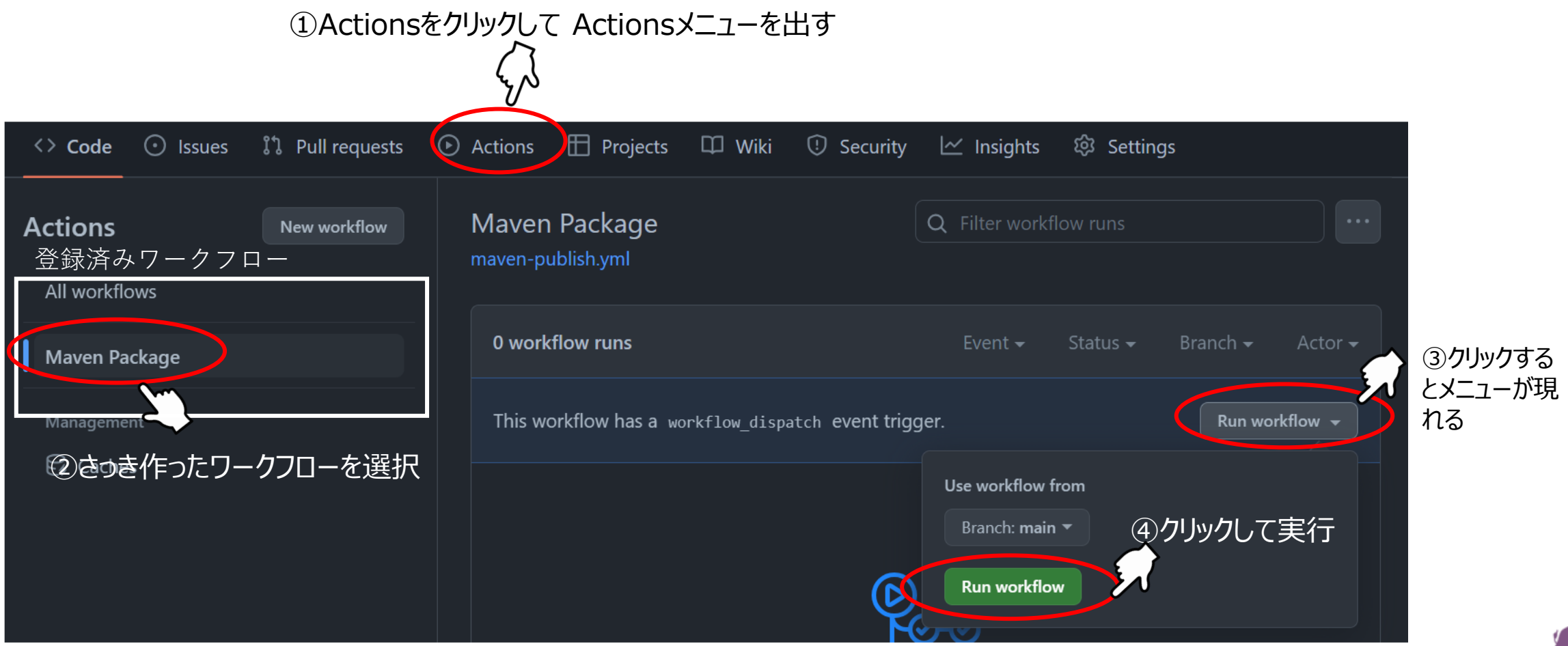
上記が追加できたら
”Commit changes..”ボタンをクリック
左のダイアログがでるので、そのまま
”Commit changes”ボタンをクリック

commitが完了すると参照画面になる

MAMEZOU-TECH

Step3. 実行してみる（失敗する） - 実行

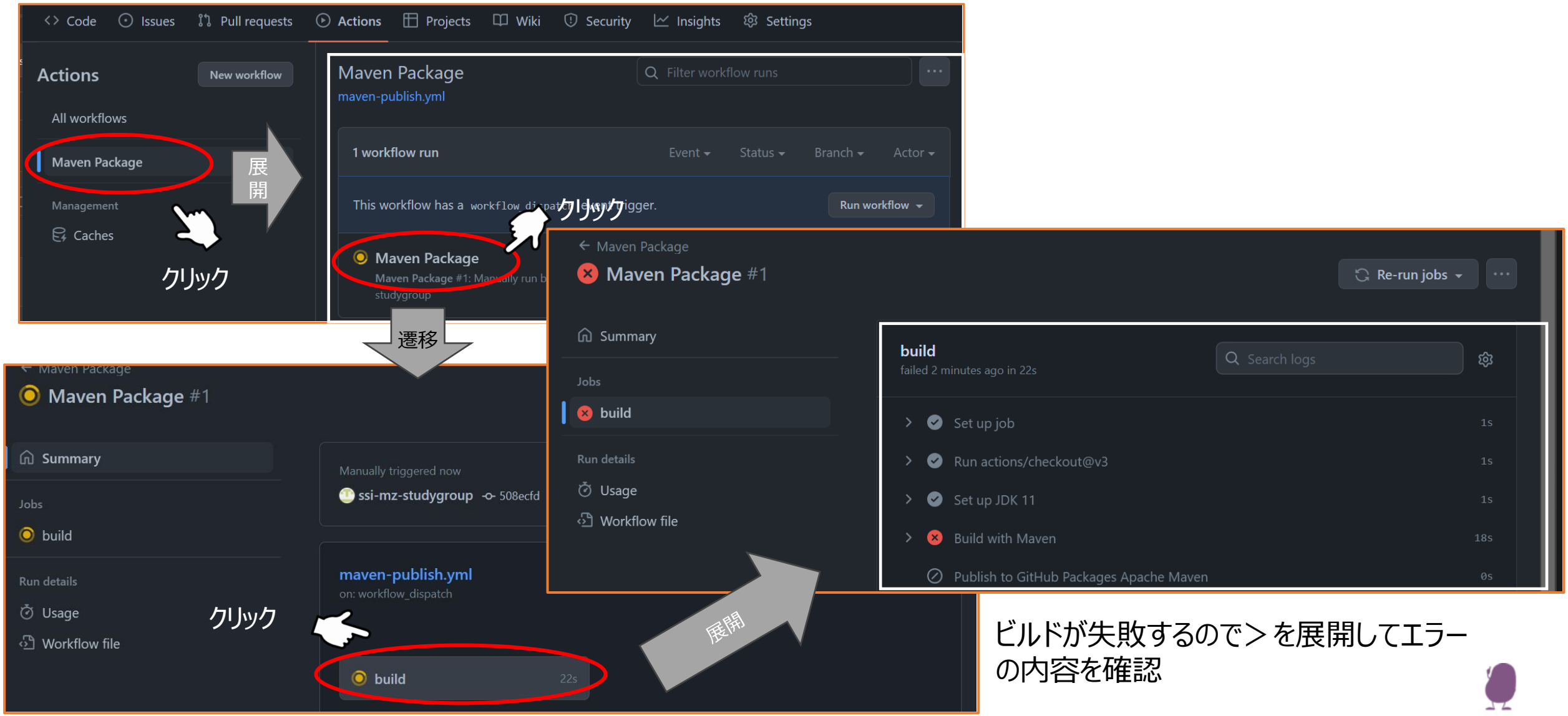
- 作成したワークフローをGitHub Actionsで実行する



※登録済みワークフローは./github/workflow以下のファイルが自動で認識される

MAMEZOU-TECH

Step3. 実行してみる（失敗する） - 状況の確認



ビルドが失敗するので＞を展開してエラーの内容を確認

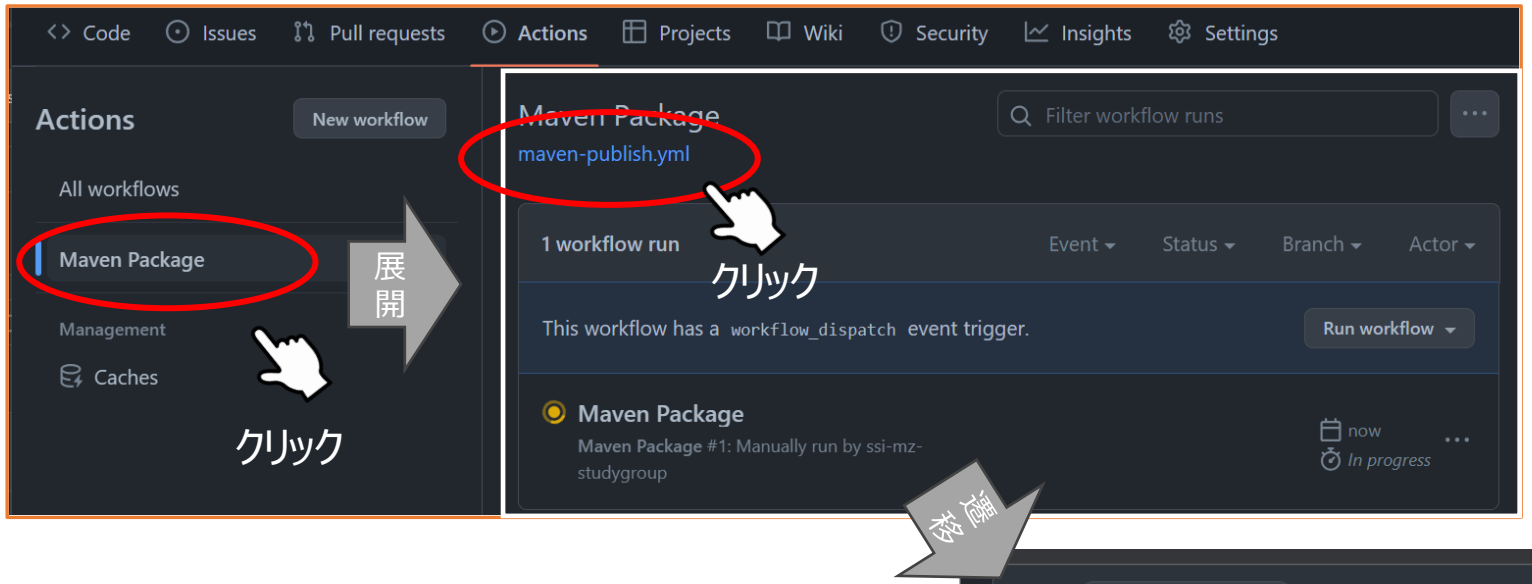
MAMEZOU-TECH

Step4. 課題を対応する

- 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
    - これもグルーグル検索すれば沢山情報があると思います

Step4. 課題を対応する（web画面からの修正）

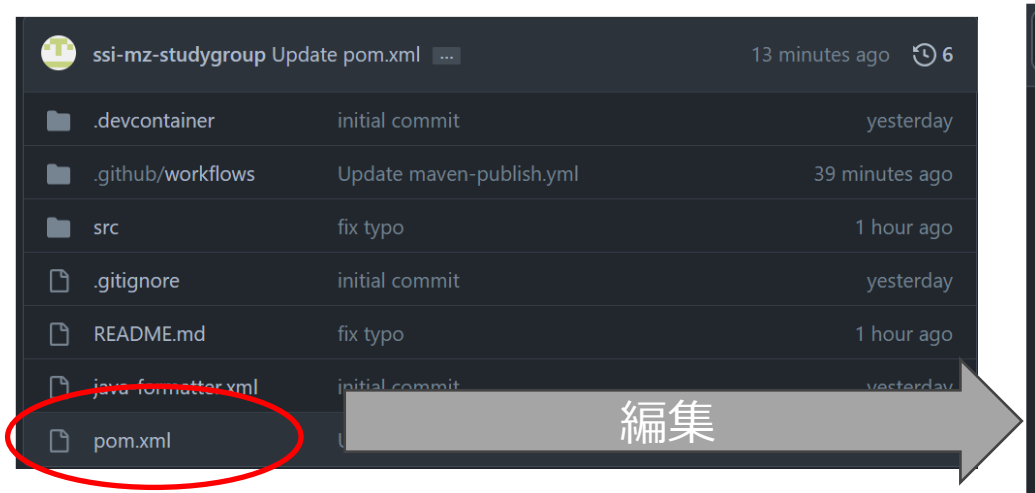
ワークフロー選択画面から



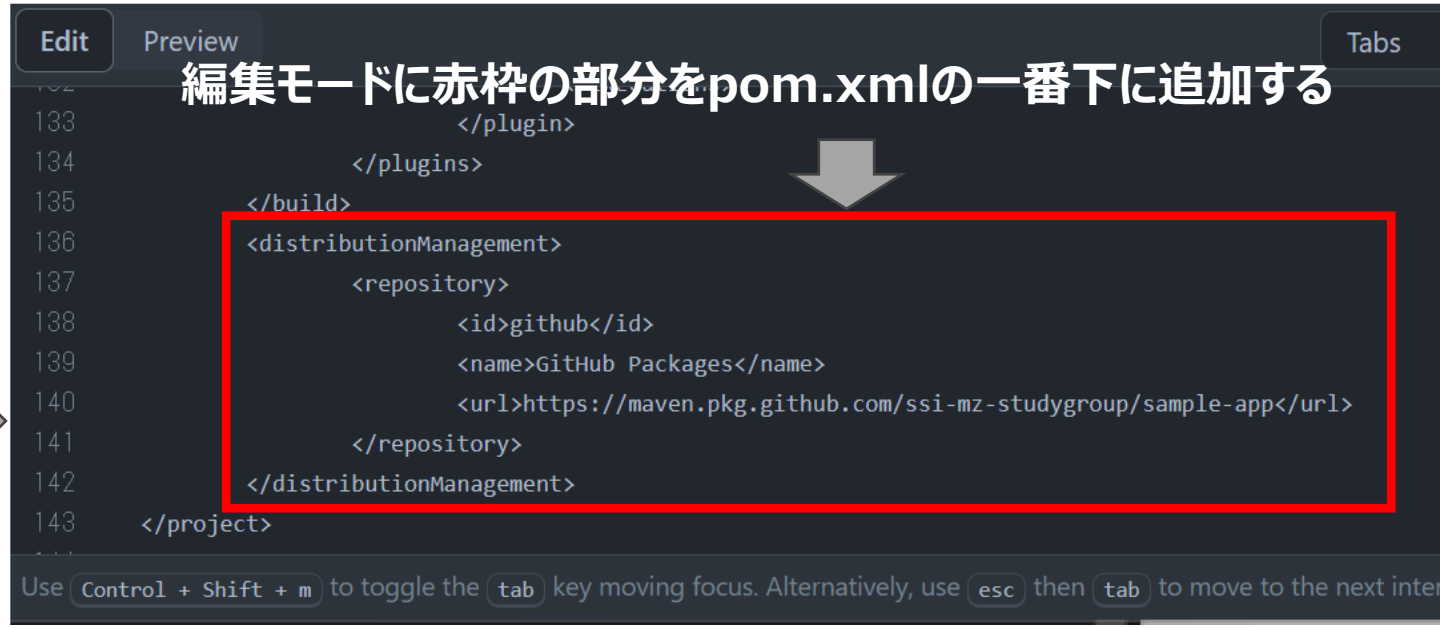
編集した後のcommit方法は「Step2. 起動条件に手動実行を追加してコミットする」で説明した手順と同じ

Step4. pom.xmlへの追加設定 – jarのデプロイ先の指定

- 今回の課題ではjarをGitHub Packagesへデプロイするため、pom.xmlへでデプロイ先としてGitHub Packagesを指定する
  - GitHub PackagesはGitHubのPackageリポジトリ機能
  - ユーザごとに専用で使えるインハウスリポジトリのようなもの



※編集モード等の操作は前スライド「Step4. 課題を対応する（web画面からの修正）」と同じ



ひな形プロジェクトのREADMEの下記コード断片(snippet)から追加する部分のひな形をコピーする
TODO:リンク

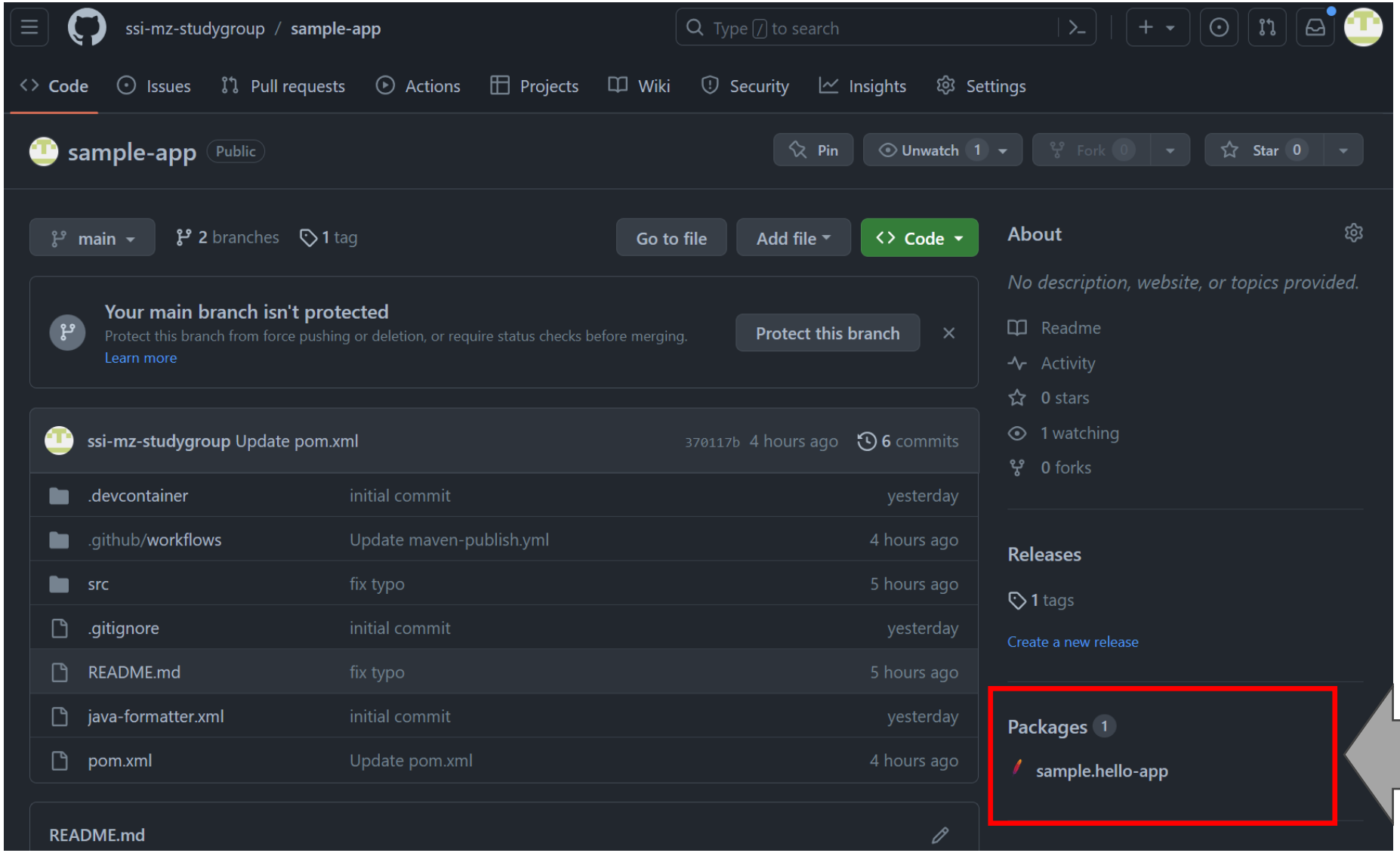
コード断片の@your_account@と@your_repository@の部分は自分のアカウントとリポジトリ名に置き換える
例）リポジトリのURLがhttps://github.com/ssi-mz-studygroup/sample-app/の場合、上のキャプチャの例のようにアカウントはssi-mz-studygroupでリポジトリ名はsample-appとなる

Step5. 実行してみる（成功するまで行う）

- 「Step3. 実行してみる」の手順でワークフローを実行し、課題を満たすまで修正＆実行を繰り返す

MAMEZOU-TECH

Step5. 実行してみる（成功するまで行う）
- GitHub Packagesへjarが登録されているかの確認 -

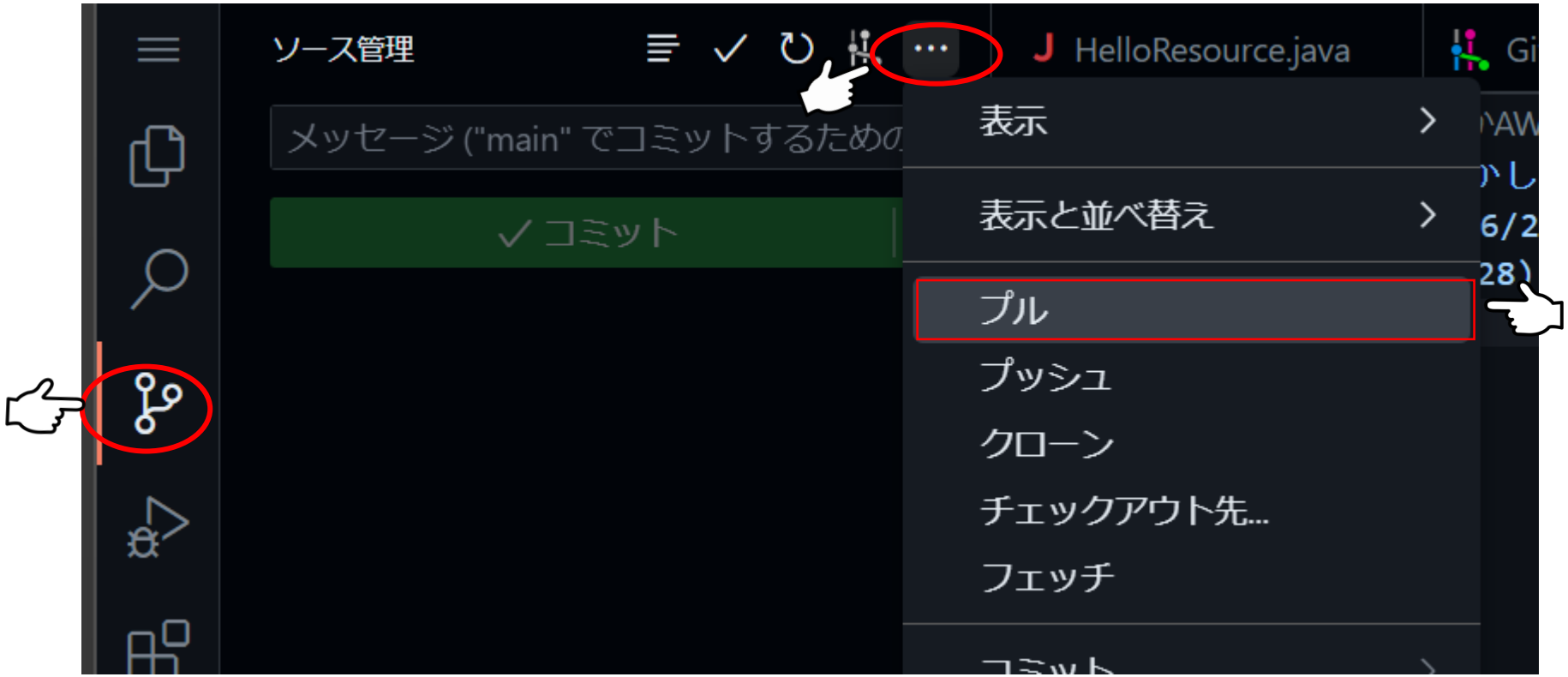


ココがあればOK

MAMEZOU-TECH

参考：Codepsacesで作業する際は

- CodespacesのLocalリポジトリにはワークフロー定義がコミットされた最新の情報が反映されていないので、Remoteリポジトリの最新を取得(pull)する

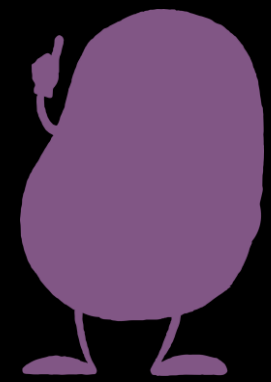


MAMEZOU-TECH

参考情報

- MavenでのJavaのビルドとテスト - GitHub Docs
  - https://docs.github.com/ja/actions/automating-builds-and-tests/building-and-testing-java-with-maven
- MavenでのJavaのパッケージの公開 - GitHub Docs
  - https://docs.github.com/ja/actions/publishing-packages/publishing-java-packages-with-maven#github-packages%E3%81%B8%E3%81%AE%E3%83%91%E3%83%83%E3%82%B1%E3%83%BC%E3%82%B8%E3%81%AE%E5%85%AC%E9%96%8B

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でビルドからテストしたものをDockerビルドしてGitHubのコンテナレジストリにデプロイ