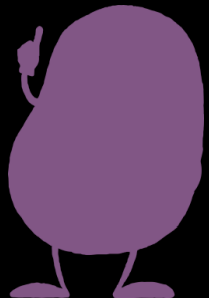


AWSとGitHubを使ってみよう勉強会

～ 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



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 -



```
src > main > java > sample > J HelloResource.java > ...
5  import jakarta.ws.rs.Path;
6  import jakarta.ws.rs.Produces;
7  import jakarta.ws.rs.core.MediaType;
8
9  @ApplicationScoped
10 @Path("hello")
11 public class HelloResource {
12     @GET
13     @Produces(MediaType.TEXT_PLAIN)
14     public String hello() {
15         return null; // TODO テストが通るように実装する
16     }
17 }
18
```

テストコードを見て修正

前回の課題

- テストコード側をしてみる -

<テストコード>

```
@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>

```
@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"

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

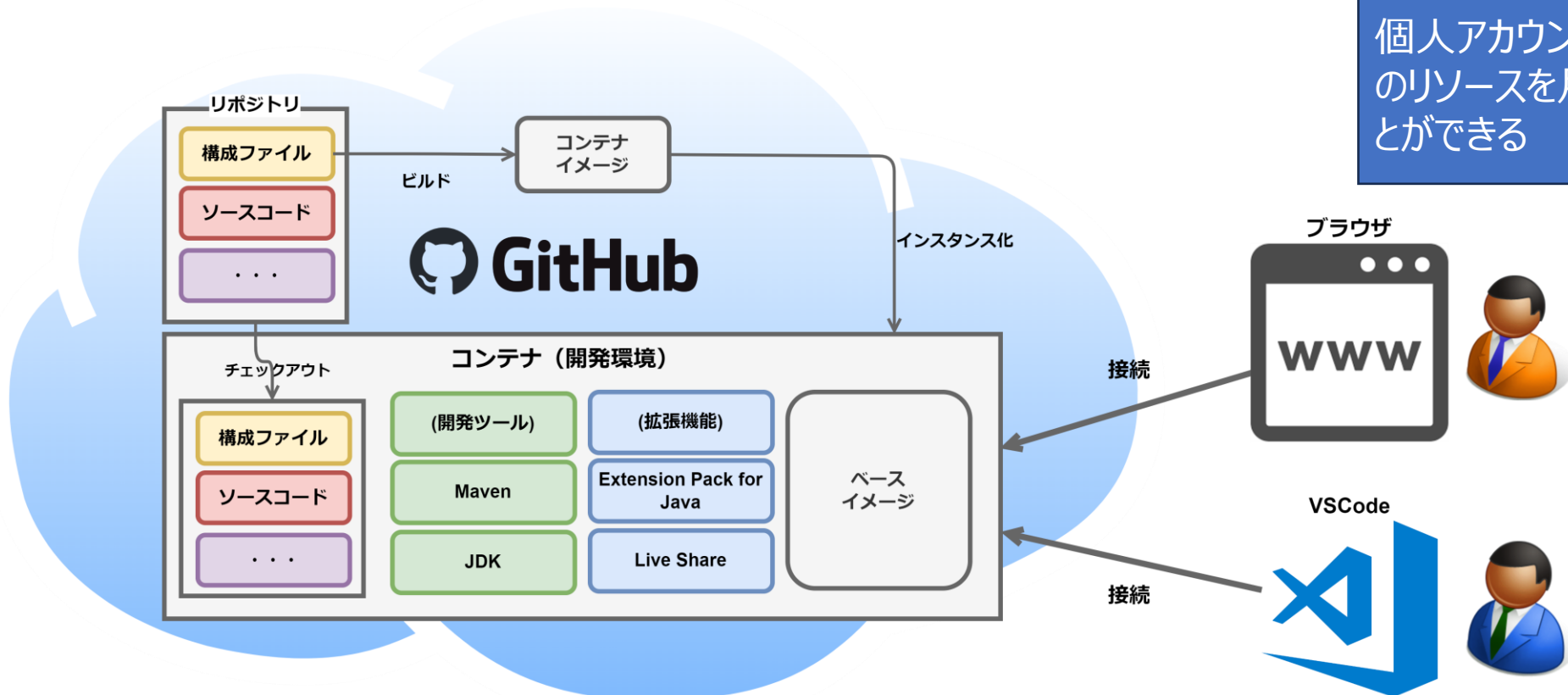
課題の補足

- 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ーに提供する機能
- ユーザー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
- ユーザー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 ]-----
```

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 default
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 available
2023-07-08 06:08:06,081 [INFO ] [o.j.w.Event] [main] - WELD-000411: Observer method [BackedAnnotatedMethod]
```

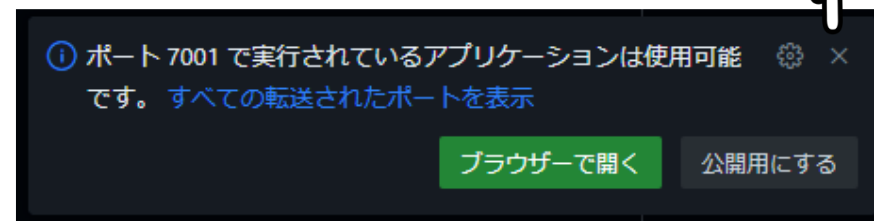
ここで紹介するコマンドは

<https://github.com/ssi-mz-studygroup/sample-app>

のREADMEにも記載しています

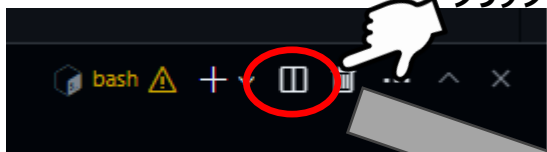
とりあえず
無視して閉じる

案内が出てくるので



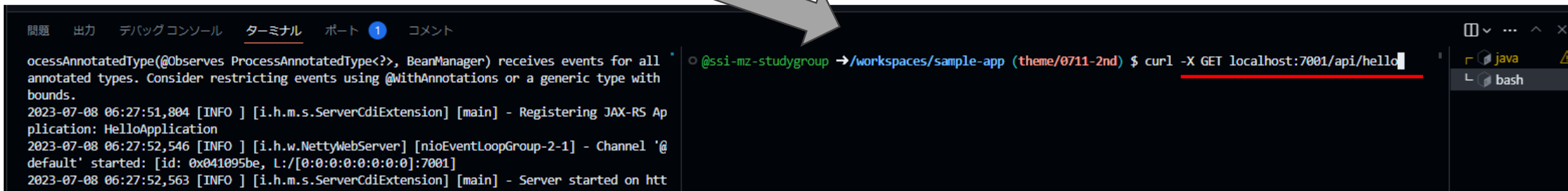
Sampleを動かしてみる(2/2)

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



Sampleが動作している環境

GitHub

コンテナインスタンス (Codepsace) 内



```
src > main > java > sample > HelloApplication.java > HelloApplication
1 package sample;
2
3 import java.util.Set;
4
5 import jakarta.enterprise.context.ApplicationScoped;
6 import jakarta.ws.rs.ApplicationPath;
7 import jakarta.ws.rs.core.Application;
8
9 @ApplicationScoped
10 @ApplicationPath("/api")
11 public class HelloApplication extends Application {
12     @Override
13     public Set<Class<?>> getClasses() {
14         return Set.of(HelloResource.class);
15     }
16 }
17
```

問題 出力 デバッグコンソール ターミナル ポート コメント

```
AnnotatedType<>, BeanManager) receives events for all annotated types. Consider restricting events using @WithAnnotations or a generic type with bounds.
2023-07-08 06:08:06,705 [INFO] [i.h.m.s.ServerCdiExtension] [main] - Registering JAX-RS Application
2023-07-08 06:08:07,453 [INFO] [i.h.w.netty.WebServer] [nioEventLoopGroup-2-1] - Channel 'default' started: [id: 0x45d1284f, L:/[0:0:0:0:0:0:0:0]:7001]
2023-07-08 06:08:07,470 [INFO] [i.h.m.s.ServerCdiExtension] [main] - Server started on http://localhost:7001 (and all other host addresses) in 2751 milliseconds (since JVM startup).
2023-07-08 06:08:07,485 [INFO] [i.h.c.HelidonFeatures] [features-thread] - Helidon MP 3.2.1 features: [CDI, Config, JAX-RS, Server]
```

起動Javaプロセス

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い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いく

つまるところ...

JakartaEEをベースとしたMSA向けAPIセット

MicroProfile 5.0

MicroProfile origin spec

Jakarta EE origin spec

Open
Tracing 3.0

Open API 3.0

RestClient 3.0

Config 3.0

Fault
Tolerance 4.0

Metrics 4.0

JWT Auth 2.0

Health 4.0

CDI 3.0

JSON-P 2.0

JAX-RS 3.0

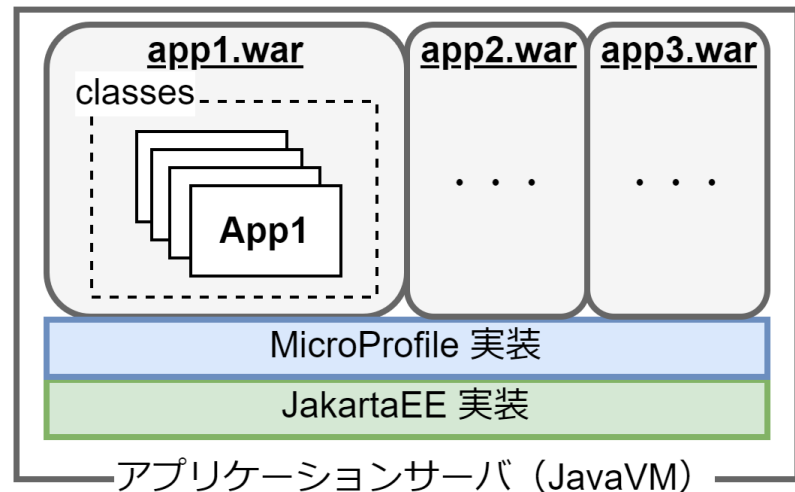
JSON-B 2.0

Jakarta
Annotations 2.0

MicroProfileの実装形態

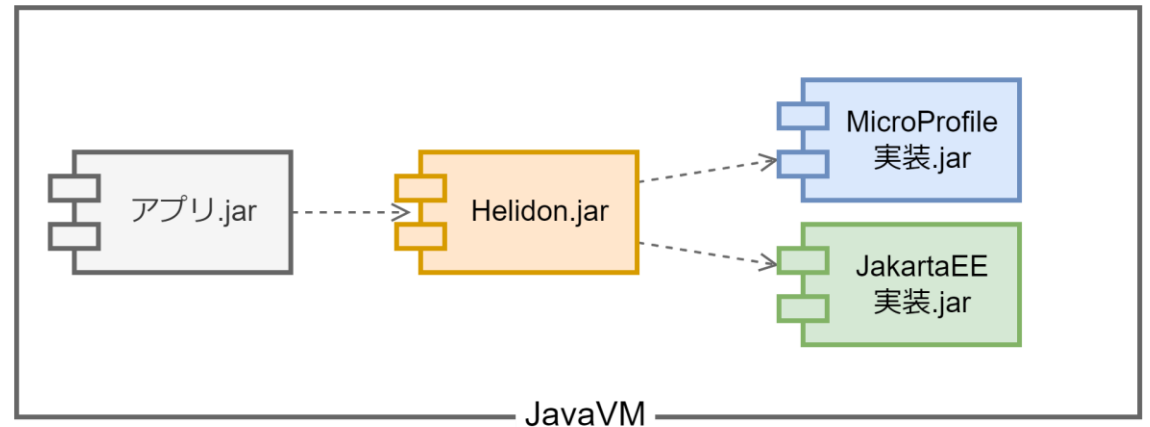
■ 2つの実装形態がある

<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)>

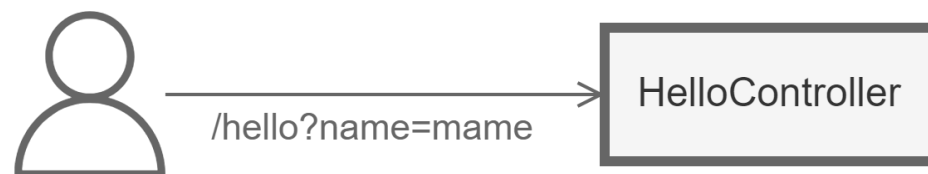


- ✓ jarのライブラリ形式で提供されるMicroProfile実装をアプリケーションのdependencyに追加して利用する
- ✓ nettyなどの組み込みのHTTPサーバを内包しているため、アプリケーションサーバは不要で動作する
- ✓ 利用イメージとしては組み込みTomcatを使ったSpringBootと同じ

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>

```

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

```

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)>

```

<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

```

<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ExecutableJar
├─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
│   ...
```

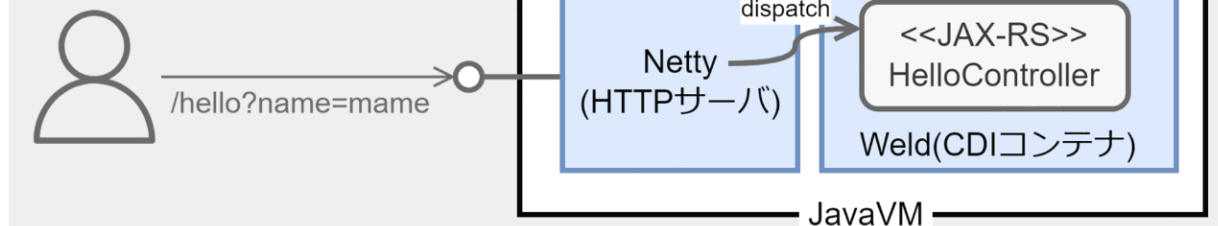
maven-dependency-pluginの
copy-dependenciesゴールで
runtimeとcompileスコープのjarが
./target/libにコピーされる

※target/libをls するとあります

2. javaコマンドでアプリを起動

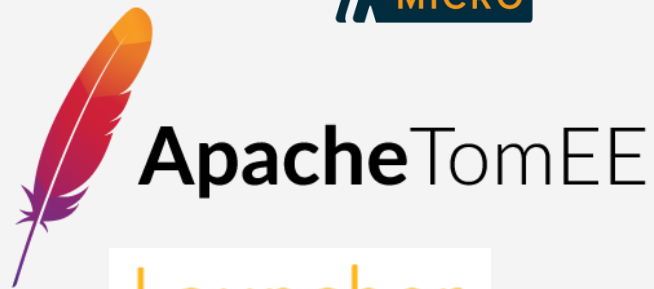
```
$ java -jar target/jjug-mp-sample.jar
```

3. 起動イメージ



主なMicroProfile実装

JakartaEEアプリケーションサーバ(warデプロイ形式)



MicroProfileフレームワーク



helidon.io



IBM陣営



Red Hat

QUARKUS

- ✓ 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

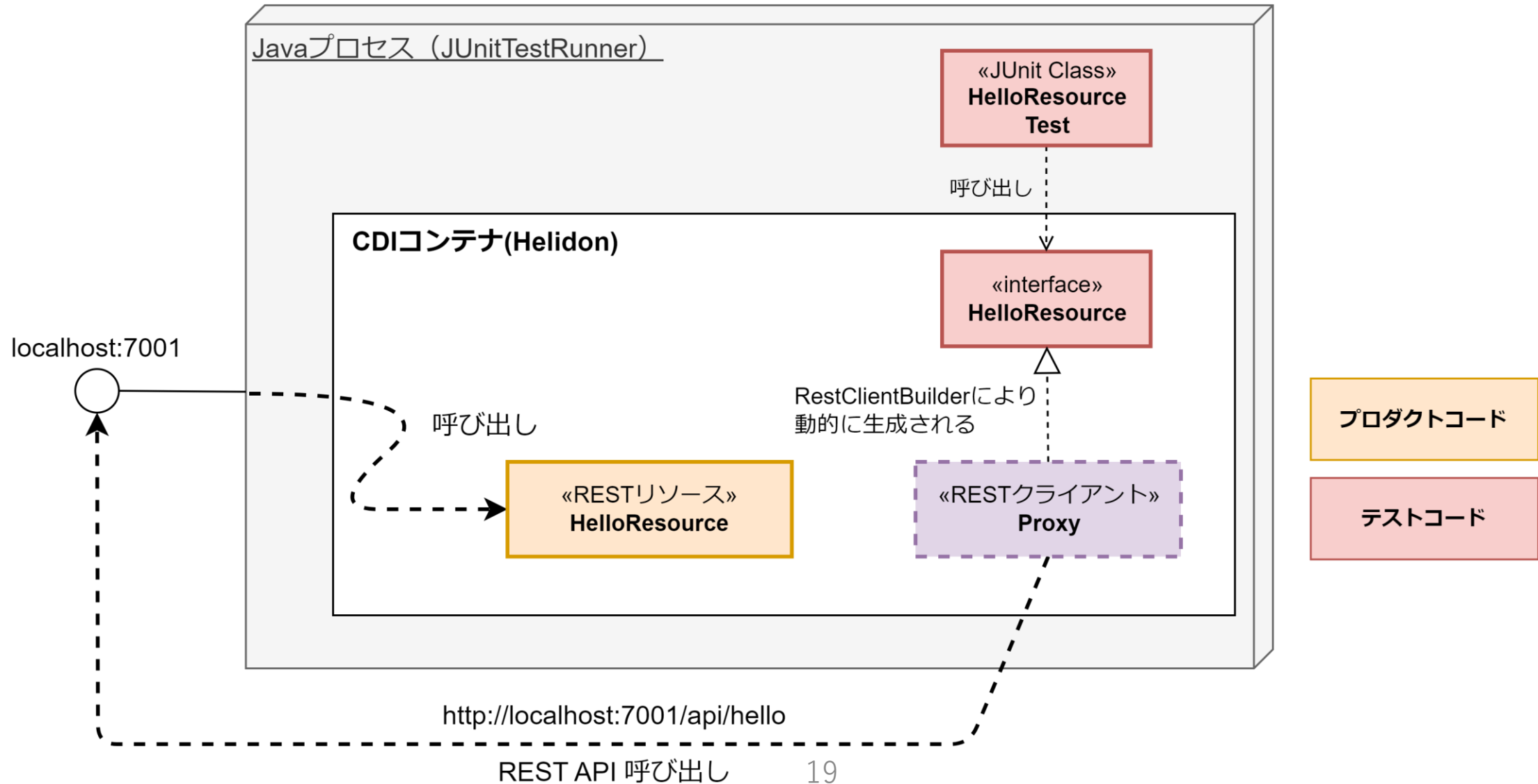
- GitHub Codespacesとは
- Sampleで使っているHelidonの仕組み
- Sampleのテストの仕組み
- Gitの補足



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



単体テストの仕組み – テストコード

```
@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 HelloResource#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

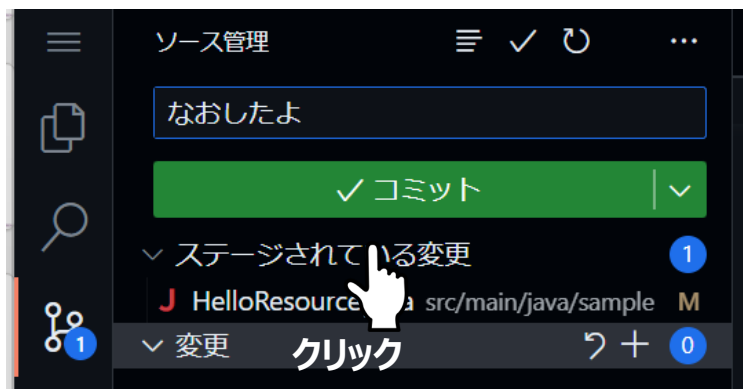
GitとSubversionの違い

- 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

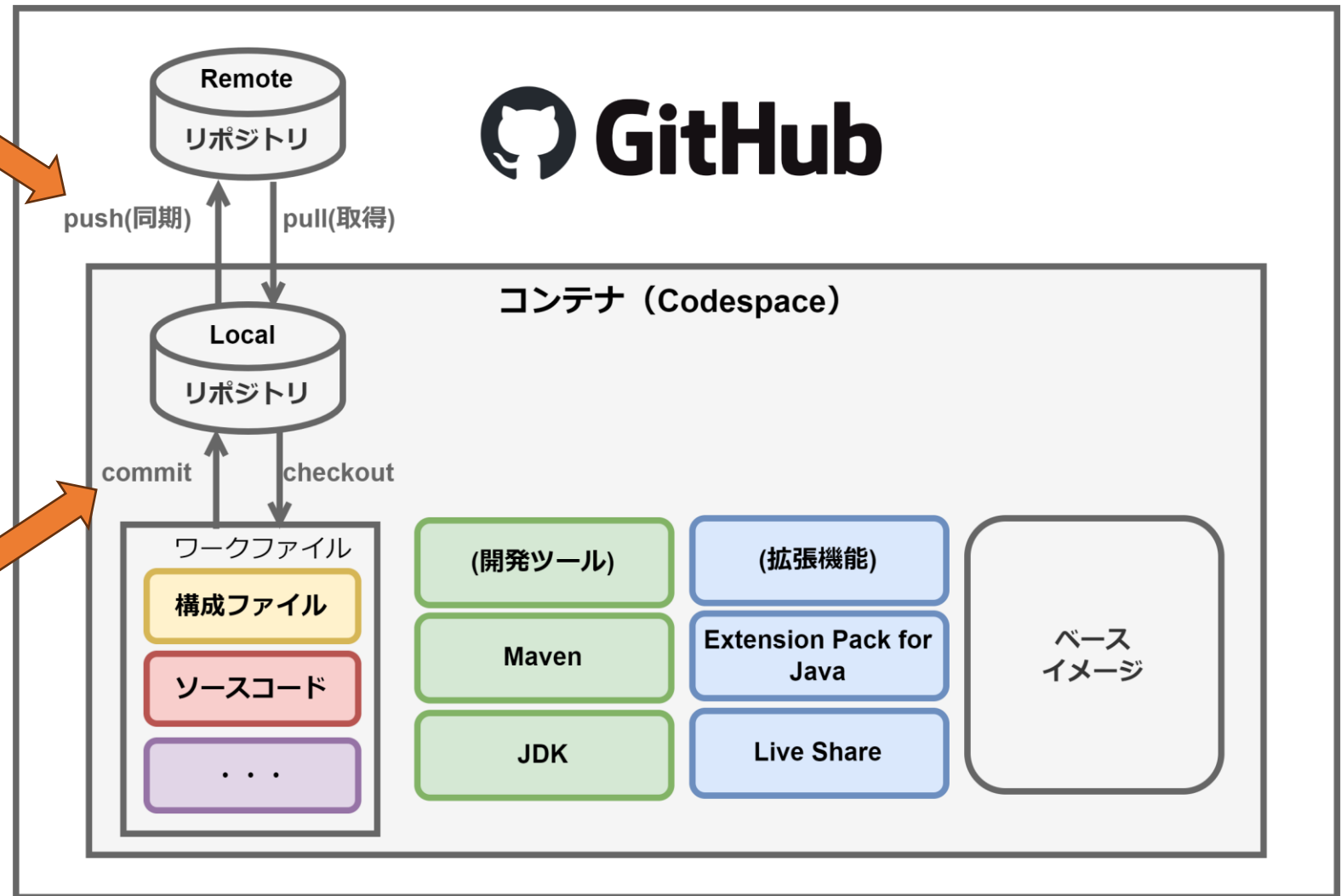
CodespacesとGitの操作の関係



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

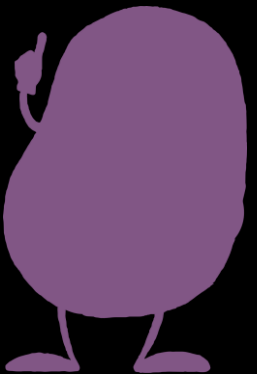
Subversionの方が優れている点

- 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。

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

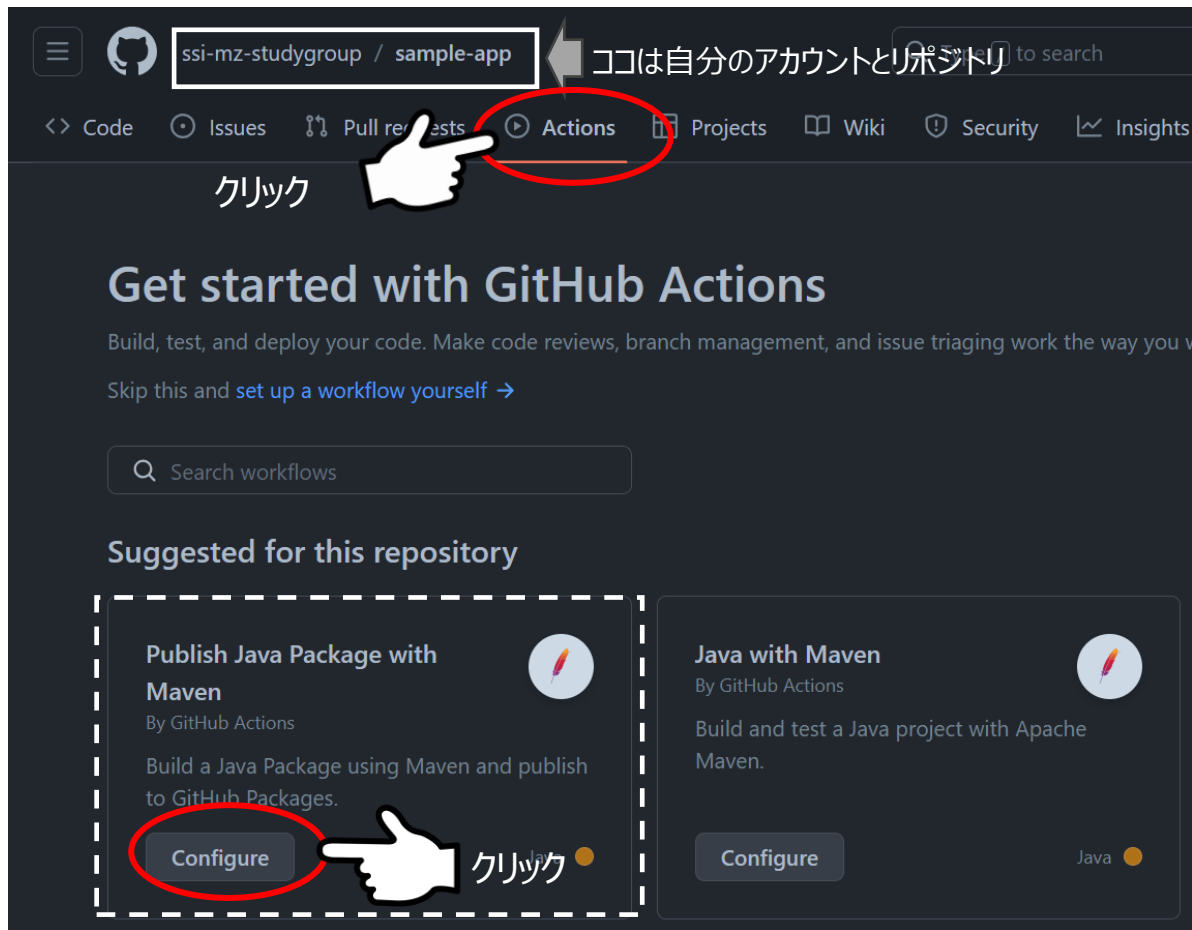
Step3. 実行してみる（失敗する）

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

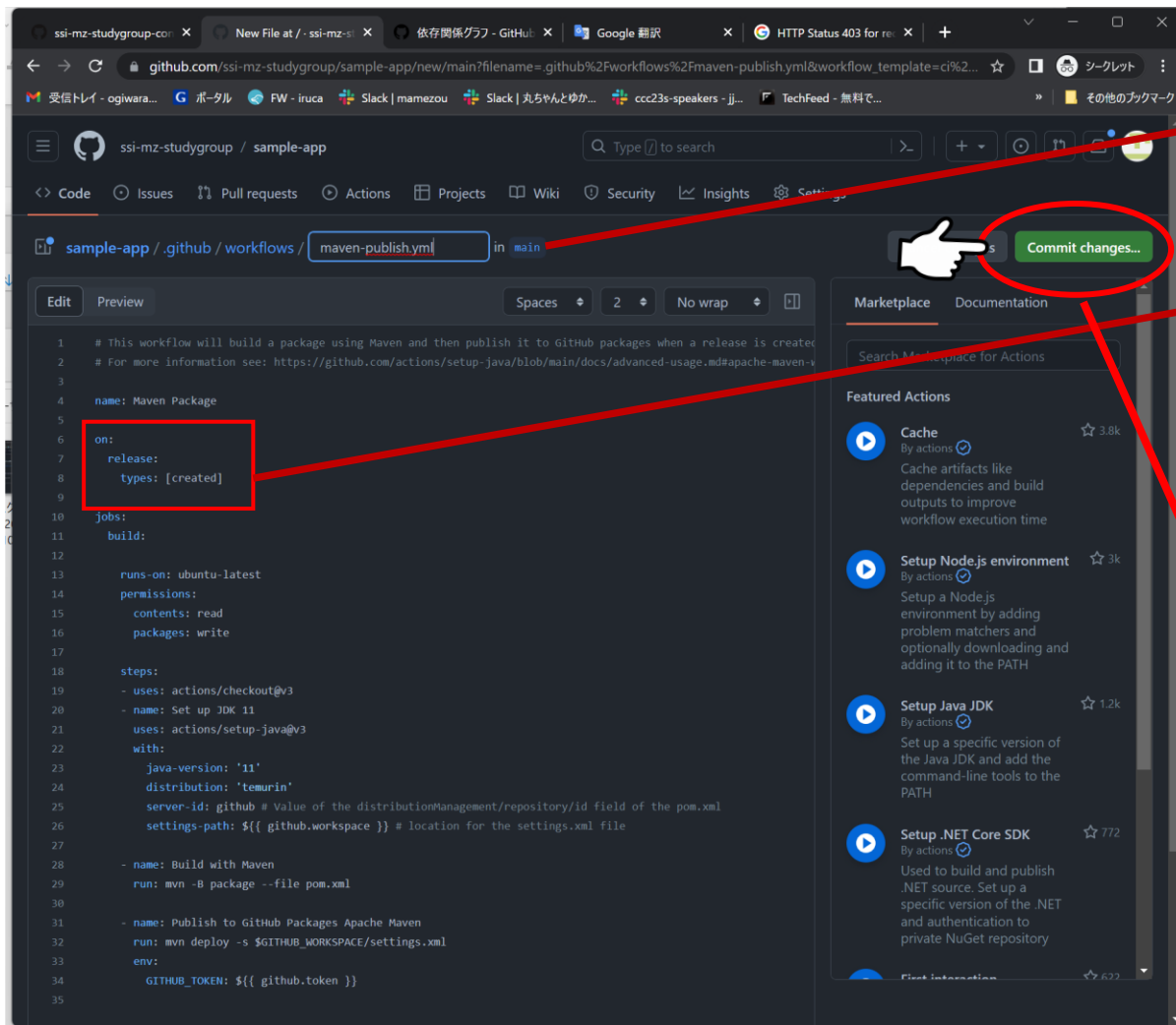


テンプレート選択画面から

- ・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

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

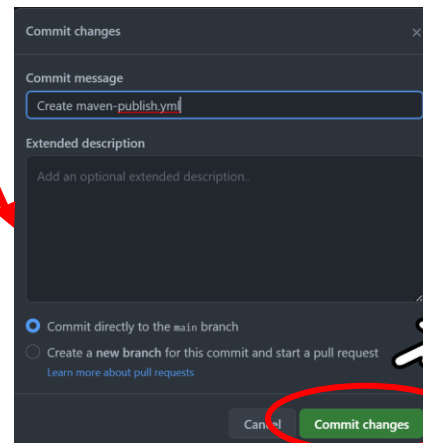
sample-app / .github / workflows / maven-publish.yml in main

ワークフロー定義のファイル名の指定⇒そのまま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

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

① Actionsをクリックして Actionsメニューを出す

The screenshot shows the GitHub Actions interface. The 'Actions' tab is selected in the top navigation bar. In the left sidebar, under 'All workflows', the 'Maven Package' workflow is highlighted with a red circle and a hand icon pointing to it. The main area shows the 'Maven Package' workflow details, including a search bar for 'Filter workflow runs' and a table with 0 workflow runs. A 'Run workflow' button is highlighted with a red circle and a hand icon pointing to it. Below this, a dropdown menu is open, showing 'Use workflow from' with 'Branch: main' selected, and a 'Run workflow' button is highlighted with a red circle and a hand icon pointing to it.

③ クリックするとメニューが現れる

④ クリックして実行

※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

Step3. 実行してみる（失敗する） - 状況の確認

The image illustrates the steps to run and check the status of a GitHub Actions workflow:

- Screenshot 1:** The 'Actions' tab is selected. The 'Maven Package' workflow is highlighted with a red circle. A hand icon points to it with the label 'クリック' (Click). A large arrow labeled '展開' (Expand) points to the next screenshot.
- Screenshot 2:** The 'Maven Package' workflow run is shown. The 'Maven Package' workflow is highlighted with a red circle. A hand icon points to it with the label 'クリック' (Click). A large arrow labeled '遷移' (Move) points to the next screenshot.
- Screenshot 3:** The 'Maven Package #1' run details are shown. The 'build' job is highlighted with a red circle. A hand icon points to it with the label 'クリック' (Click). A large arrow labeled '展開' (Expand) points to the next screenshot.
- Screenshot 4:** The 'build' job details are shown. The 'Build with Maven' step is highlighted with a red circle. A hand icon points to it with the label 'クリック' (Click). A large arrow labeled '展開' (Expand) points to the next screenshot.

ビルドが失敗するので>を展開してエラーの内容を確認

Step4. 課題を対応する

- 課題その 1

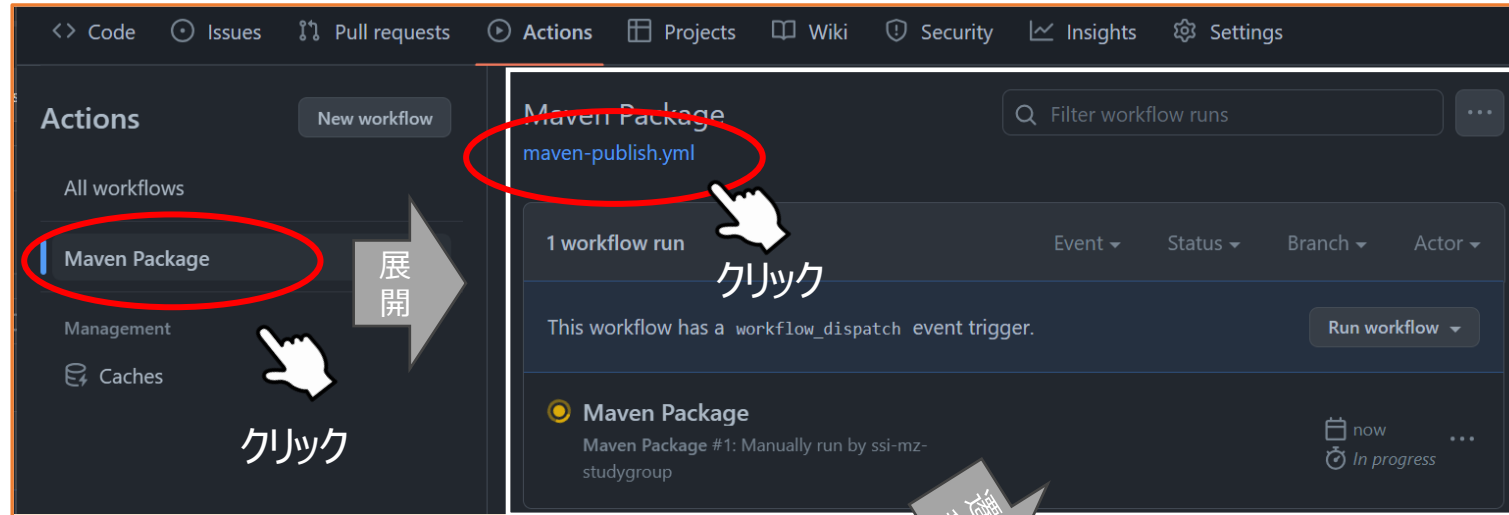
- ビルドエラーにならないようにワークフロー定義を修正する
- 後続スライドのpom.xmlへの追加設定を行う
- ヒント
 - サンプルのアプリで使っているHelidonはJava17を必要とします
 - uses: actions/setup-java@v3のようにusesプロパティの右側にあるものはJP1の組み込みJOBのようなものに相当します。なので、何者か？やどんなプロパティ設定があるのか？などは“actions/setup-java”で検索すると分かると思います

- 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

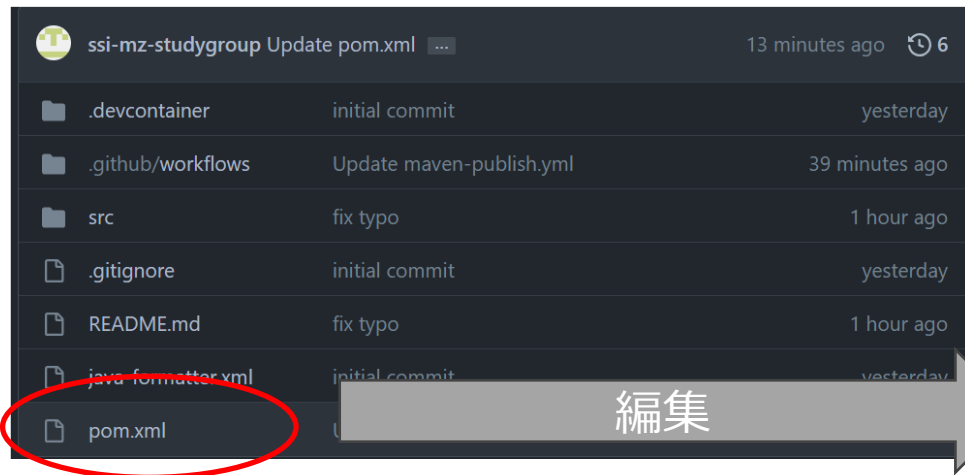


編集した後のcommit方法は「Step2. 起動条件に手動実行を追加してコミットする」で説明した手順と同じ



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シ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



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 -

The screenshot shows the GitHub interface for a repository named 'sample-app' under the user 'ssi-mz-studygroup'. The repository is public and has 2 branches and 1 tag. The main branch is 'main'. A notification states 'Your main branch isn't protected'. The file list shows several files, including 'pom.xml', which was updated 4 hours ago. On the right side, the 'About' section is empty, and the 'Releases' section shows 1 tag. The 'Packages' section is highlighted with a red box and contains one package named 'sample.hello-app'.

ssi-mz-studygroup / sample-app

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

sample-app Public

main 2 branches 1 tag

Go to file Add file Code

Your main branch isn't protected

Protect this branch

ssi-mz-studygroup Update pom.xml 370117b 4 hours ago 6 commits

File	Commit	Time
.devcontainer	initial commit	yesterday
.github/workflows	Update maven-publish.yml	4 hours ago
src	fix typo	5 hours ago
.gitignore	initial commit	yesterday
README.md	fix typo	5 hours ago
java-formatter.xml	initial commit	yesterday
pom.xml	Update pom.xml	4 hours ago

Releases

1 tags

Create a new release

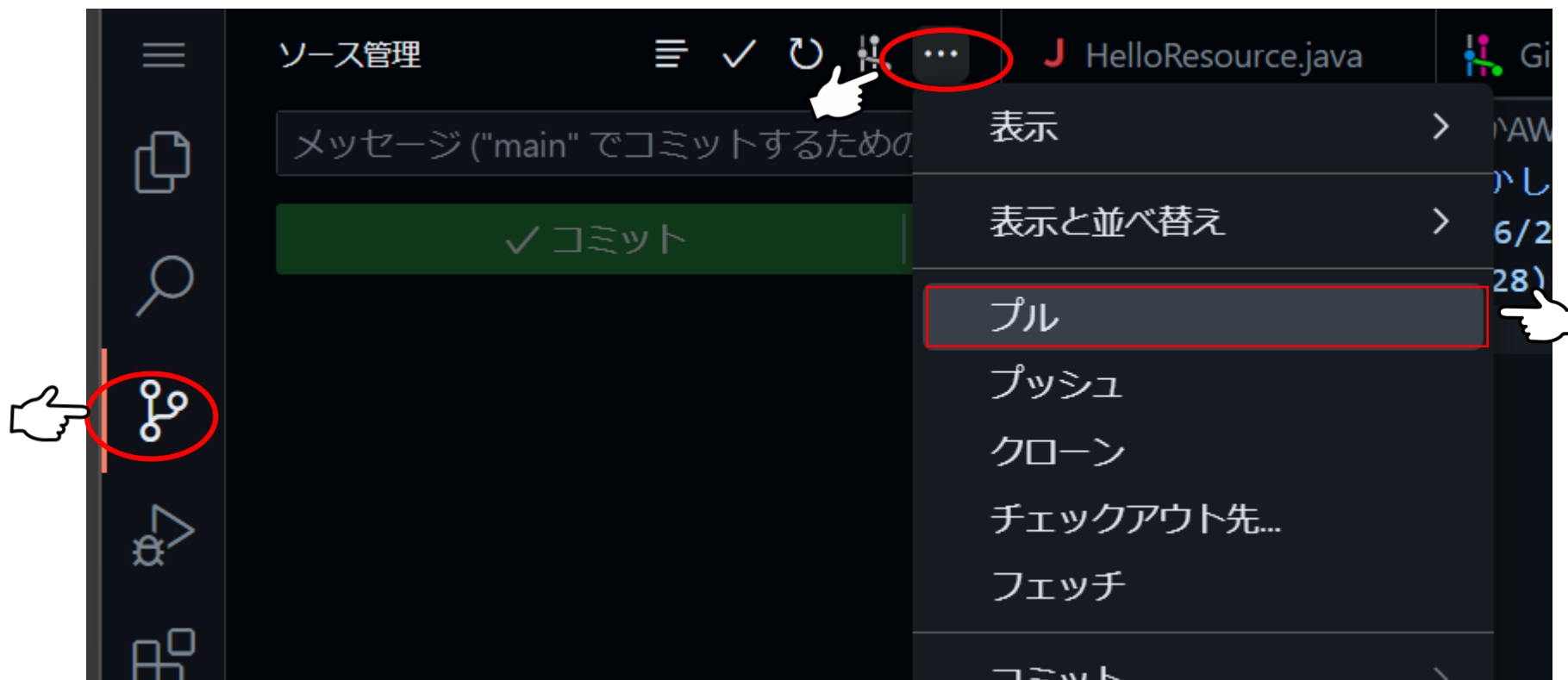
Packages 1

sample.hello-app

ココがあればOK

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



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～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