

Web アプリケーション入門

目次

[1. Web アプリケーションとは](#)

[2. フレームワーク](#)

[2.1 利点](#)

[2.2 選択](#)

[3. Spring Boot](#)

[3.1 Spring framework](#)

[3.2 Spring Boot](#)

[3.3 プログラミング言語](#)

[3.4 開発環境](#)

[3.5 Spring Initializr](#)

[3.6 Spring Boot のバージョン変更](#)

[3.6.1 変更手順](#)

[3.6.2 注意点 :](#)

[3.6.3 他の方法 :](#)

[3.7 依存性の更新](#)

[4. 簡単な Web アプリケーション](#)

[4.1 \(手順 1\) 初期プロジェクトを作成](#)

[4.2 \(手順 2\) 初期プロジェクトをダウンロード](#)

[4.3 \(手順 3\) 初期プロジェクトを Eclipse にインポート](#)

[4.4 \(手順 4\) 初期プロジェクトをカスタマイズ](#)

[4.4.1 カスタマイズの場所](#)

[4.4.2 カスタマイズの内容](#)

[4.5 Web アプリケーションを起動](#)

[4.6 結果を確認](#)

[4.7 Web アプリケーションを停止](#)

[5. Web アプリケーションの設計と構造](#)

[5.1 アプリケーションアーキテクチャ](#)

[5.2 MVC \(Model View Controller\)](#)

[5.3 レイヤードアーキテクチャ](#)

[5.4 クリーンアーキテクチャ](#)

[6. Java パッケージ構成の例](#)

[6.1 層型アーキテクチャ](#)

6.2 レイヤーアーキテクチャ的な例

6.3 クリーンアーキテクチャ的な例

7. 取引銀行管理システム（サンプルアプリ）

7.1 アプリケーションの要件

7.2 ユースケース

7.3 プロジェクト作成時の手順

7.3.1 Initializr で初期プロジェクトを作成

7.3.2 GitHub でリポジトリ mainbank を作成（任意：必要な方のみ）

7.4 ドメインモデルと永続化（継承の永続化戦略）

8. Spring 依存性注入とは

8.1 DI の基本的な考え方

8.2 DI の例

8.3 DI の仕組み

8.4 DI のメリット

8.5 依存性とは

8.6 注入とは

9. Spring アノテーション

9.1 主要なアノテーション

9.2 バリデーション関係

9.3 JPA 関係

10. コントローラとビューの間のデータの受け渡し

10.1 View → Controller

10.1.1 補足説明

10.2 Controller → View

11. Spring アスペクト指向プログラミング

11.1 構成要素

11.1.1 アスペクト (Aspect)

11.1.2 アドバイス (Advice)

11.1.3 ジョインポイント (Join Point)

11.1.4 ポイントカット (Pointcut)

11.1.5 ターゲット (Target)

11.2 簡単な例

12. Spring JPA 永続化

12.1 エンティティオブジェクトの状態遷移

12.2 エンティティオブジェクトの状態

[12.3 トランザクション境界での状態遷移の挙動](#)

[13. MySQL](#)

[13.1 データベースの作成](#)

[13.2 ユーザとパスワードの作成](#)

[13.2.1 username と password の確認方法と設定方法](#)

[13.3 application.properties の設定](#)

[13.4 データベースとスキーマ](#)

[13.5 spring.jpa.hibernate.ddl-auto の設定](#)

[13.5.1 代表的なオプション](#)

[13.5.2 オプションのまとめ](#)

[14. Tips レンダリング](#)

[14.1 サーバサイドレンダリング \(Server-Side Rendering, SSR\)](#)

[14.2 クライアントサイドレンダリング \(Client-Side Rendering, CSR\)](#)

[14.3 静的サイト生成 \(Static Site Generation, SSG\)](#)

[15. Tips : セレクトボックスの書き方](#)

1. Web アプリケーションとは

Web アプリケーション (Web Application、Web App) とは、インターネットを介して、Web ブラウザ上で動作するアプリケーションのことです。

例 マイナポータル (行政手続のオンライン窓口)



2. フレームワーク

マイナポータルなど、さまざまな Web アプリケーションが存在しますが、それらがゼロから全て開発されているわけではありません。通常、フレームワークと呼ばれるソフトウェアを基盤として開発が行われます。Web アプリケーション用フレームワークとは、多くの Web アプリケーションで共通して利用できる構造や基本的な機能を提供する枠組みです。

フレームワークには、PHP 用、Ruby 用、そして Java 用のものがあり、Java では特に Spring が広く使われています。なお、同じプログラミング言語でも複数のフレームワークが存在します。多くのフレームワークは、オープンソースソフトウェアとして無料で利用することが可能です。

フレームワークを使うことで、開発者は基礎的な部分を一から作る手間を省き、アプリケーションの本質的な機能の開発に集中することができます。

2.1 利点

フレームワークを使用すると次のような利点があります。

(1) 再利用できる部品を提供します

フレームワークには、よく使われる機能やパターン（ユーザー認証、データベース接続、ページ遷移など）が組み込まれています。それを利用することで、毎回ゼロからコードを書く必要がありません。

(2) 標準的な構造を提供します

フレームワークは、アプリケーションの構造や作り方にある程度のルールを提供します。このため、どの開発者も一貫した方法でコードを書くことができ、チームでの開発が容易になります。

(3) セキュリティやパフォーマンスを向上できます

フレームワークには、セキュリティやパフォーマンスに関する考慮がされており、これを利用することでセキュアで効率的な Web アプリケーションを作ることができます。

2.2 選択

Web アプリケーションを作成する際には、何らかのフレームワークを使用することが一般的です。フレームワークは、使用するプログラミング言語によって選択肢が決まってきます。

Java 言語用としては、本書で紹介する Spring の他にも Apache Struts などがあります。

Python 用の場合は、Django や Flask、PHP 用の場合は、Laravel や CakePHP、Ruby 用の場合は、Ruby on Rails、C# の場合は、ASP.NET Core Blazor などがあります。

3. Spring Boot

Spring Boot は Spring フレームワークの一部分（サブセット）です。Spring Boot について説明する前に、Spring について説明します。

3.1 Spring framework

Spring framework（以降 Spring と略称します）は、Java でエンタープライズアプリケーションを開発するためのフレームワークです。主な特徴は、依存性注入（DI）やアスペクト指向プログラミング（AOP）などを使って、柔軟でスケーラブルなアプリケーションを構築しやすくする点です。Spring を使うと、ビジネスロジックに集中しやすくなります。

一方で、Spring は多くの機能を有するフレームワークであるため、どの機能を選択し使用すればよいのか戸惑うこともありました。そこでこの問題を解消するために登場したのが Spring Boot です。

3.2 Spring Boot

Spring Boot は、Spring をより簡単に使えるようにしたフレームワークです。自動設定機能や組み込み Web サーバー（Tomcat など）が提供されているため、設定が少なく、すぐにアプリケーションを起動できるのが特徴です。スムーズに Spring のプロジェクトを始められる点が Spring Boot の大きなメリットです。

3.3 プログラミング言語

Spring フレームワークは、主に Java をサポートしていますが、他の複数のプログラミング言語も利用可能です。以下の言語が Spring フレームワークおよび Spring Boot でサポートされています：

- Java

Spring は元々 Java のために設計されたフレームワークであり、Java が主な言語です。

- Kotlin

Spring フレームワークおよび Spring Boot では、Kotlin も公式にサポートされています。

- Groovy

Groovy は、動的言語でありながら JVM 上で動作するスクリプト言語です。

これらの言語はすべて Spring Boot でも同様にサポートされており、Java プロジェクトと同じように、Kotlin や Groovy で Spring Boot アプリケーションを構築することができます。

例えば、Spring Initializr（後述）を使って、Kotlin や Groovy ベースの Spring Boot プロジェクトを作成することも可能です。

3.4 開発環境

本書では、統合開発環境（IDE）として「Pleiades All in One」（以降 Eclipse と記載）を使用します。

（参考）

もし、Java 言語や IDE に関する情報が必要でしたら

「Java プログラミング入門」の「1. 学習環境を準備します」を参考ください。

<http://www.fk-nextdesign.sakura.ne.jp/learn/Roadmap.html>

3.5 Spring Initializr

本書では Spring の初期プロジェクト (IDE にインポート可能な空の Spring アプリケーションプロジェクト) を作成するために、「Spring Initializr」を利用します。

Spring Boot は、Spring を使いやすくしましたが、それでも、最初に Spring Boot 用の Java プロジェクトを構成するときはボタンひとつでは済みません。そこで Web 画面から構成できるようにしたのが、Spring Initializr です。この初期プロジェクトを Eclipse にインポートして、独自のプログラムを追加・編集することで、自前の Web アプリケーションとして作成します。具体的な操作は後述します。

3.6 Spring Boot のバージョン変更

アプリケーションの開発が進んだ後で Spring Boot を最新バージョンにしたい場合があります。バージョンの変更方法を紹介します。

3.6.1 変更手順

- (1) pom.xml をエディタで開いて、バージョンを更新します。

<parent> タグ内の <version> を最新バージョンに変更。Spring Initializr のサイトで最新バージョンを確認するか、Maven Central Repository で最新のバージョンを調べて指定します。

例: <version>3.4.0</version>

- (2) Eclipse でプロジェクトを更新

Eclipse で pom.xml を右クリック → Maven → プロジェクトの更新 を選択します。これにより、Maven が新しいバージョンをダウンロードして設定を更新します。

3.6.2 注意点 :

- (1) 互換性の確認

Spring Boot のメジャーバージョンアップ (例: 2.x → 3.x) やマイナーバージョンアップ時に、依存関係や設定ファイルの互換性が問題になる場合があります。公式の Spring Boot リリースノートを確認して、変更点や注意点を事前にチェックしてください。

- (2) Java バージョンの確認

Spring Boot 3.x 系を使用する場合、最低でも Java 17 が必要です (現在の pom.xml の設定は問題ありません)。

- (3) 依存関係の更新

Spring Boot のバージョンを上げた場合、使用している依存関係も更新が必要になる場合があります。spring-boot-starter-parent が依存関係の管理をしてくれますが、場合によっては明示的にアップデートが必要です。

3.6.3 他の方法 :

方法 1: Spring Initializr を利用して新規プロジェクトを作成

最新の Spring Boot バージョンで新規プロジェクトを Spring Initializr から作成し、既存のプロジェクトのコードをコピーして移行する方法です。これにより、すべての依存関係が最新の状態になります。

方法 2: コマンドラインを使用してアップデート

Maven コマンドを使うことで、Eclipse を使わずにアップデートできます。

bash

```
mvn versions:set -DnewVersion=3.4.0
```

```
mvn clean install
```

versions:set コマンドでバージョンを変更し、clean install で依存関係を更新します。

方法 3: spring-boot-dependencies を使用して依存関係を管理

spring-boot-dependencies を直接利用することで、依存関係の整合性が保たれます。pom.xml の <parent> 部分を以下のように変更してください：

xml

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-dependencies</artifactId>

<version>3.4.0</version>

<relativePath/>

</parent>

3.7 依存性の更新

以下は、セキュリティ依存性を追加する例です

(1) SpringInitializr を開きます。

The screenshot displays the Spring Initializr configuration page. On the left, under 'Project', 'Maven' is selected. Under 'Language', 'Java' is selected. Under 'Spring Boot', '3.4.0' is selected. In the 'Project Metadata' section, 'Packaging' is set to 'Jar' and 'Java' version is set to '17'. On the right, the 'Dependencies' section shows 'Spring Security' as a selected dependency. At the bottom, the 'EXPLORE' button is highlighted with a red box.

Project

☐ Gradle - Groovy ☒ **Java** ☐ Kotlin

☐ Gradle - Kotlin ☐ Groovy

☒ **Maven**

Spring Boot

☐ 3.4.1 (SNAPSHOT) ☒ **3.4.0** ☐ 3.3.7 (SNAPSHOT) ☐ 3.3.6

Project Metadata

Group: com.example

Artifact: demo

Name: demo

Description: Demo project for Spring Boot

Package name: com.example.demo

Packaging: ☒ **Jar** ☐ War

Java: ☐ 23 ☐ 21 ☒ **17**

Dependencies

ADD DEPENDENCIES... CTRL + B

Spring Security **SECURITY**

Highly customizable authentication and access-control framework for Spring applications.

GENERATE CTRL + G **EXPLORE** CTRL + SPACE **SHARE...**

赤枠の中を、現在のプロジェクトの設定に合わせます。

右上の「ADD DEPENDENCIES...」をクリックし、

Spring Scurity のみを選択し、

画面下部の EXPLORE をクリックすると、セキュリティ依存性だけを含む pom.xml が表示されます。（次図）

（現在の pom.xml には無い）<dependency>…</dependency>を、現在の pom.xml に追加します。

Eclipse で pom.xml を右クリック → Maven → プロジェクトの更新 を選択します。

demo.zip

.gitattributes

.gitignore

.mvn

HELP.md

mvnw

mvnw.cmd

pom.xml

src

DOWNLOAD

COPY

5

<parent>

6

<groupId>org.springframework.boot</groupId>

7

<artifactId>spring-boot-starter-parent</artifactId>

8

<version>3.4.0</version>

9

<relativePath/> <!-- Lookup parent from repository -->

10

</parent>

11

<groupId>com.example</groupId>

12

<artifactId>demo</artifactId>

13

<version>0.0.1-SNAPSHOT</version>

14

<name>demo</name>

15

<description>Demo project for Spring Boot</description>

16

<url/>

17

<licenses>

18

<license/>

19

</licenses>

20

<developers>

21

<developer/>

22

</developers>

23

<scm>

24

<connection/>

25

<developerConnection/>

26

<tag/>

27

<url/>

28

</scm>

29

<properties>

30

<java.version>17</java.version>

31

</properties>

DOWNLOAD CTRL + ↵

CLOSE ESC

追加する部分

※網掛けの依存性（spring-boot-starter-test）はすでに追加済みのはずなので、追加不要です。

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

この例の場合は、赤枠内の2つの<dependency>...</dependency>を現在の pom.xml に追加します。

<dependencies>...</dependencies>の中の最後などです。

この辺はどんな依存性を追加するのかによって違ってきます。

なお、以下の部分はすべてのプロジェクトに共通するものなので追加しません。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
```

4. 簡単な Web アプリケーション

Eclipse を使って Java Spring Web アプリケーションを作成するためには、まず Eclipse 上に新しいプロジェクトを作成します。方法はいくつかありますが、本書では次の手順で作成します。

4.1 （手順1）初期プロジェクトを作成

- (1) Spring initializr にアクセスします

<https://start.spring.io/>

(2) 次図のように設定します

本サンプルでは Maven を選択しますが、他でも問題ありません。

Dependencies（依存性）は次の3つを選択します。

右上の「ADD …」ボタンを押下して Dependency を選択し追加します。

- Spring Web
- Thymeleaf
- Spring Boot DevTool

上の構成は、Thymeleaf を使う場合の最小構成です。

もし、永続化（データベース）を使用するような場合は、次の依存性が必要です。

この依存性の構成は一例です。例えば、MySQL を使用したい場合は別の構成になります。

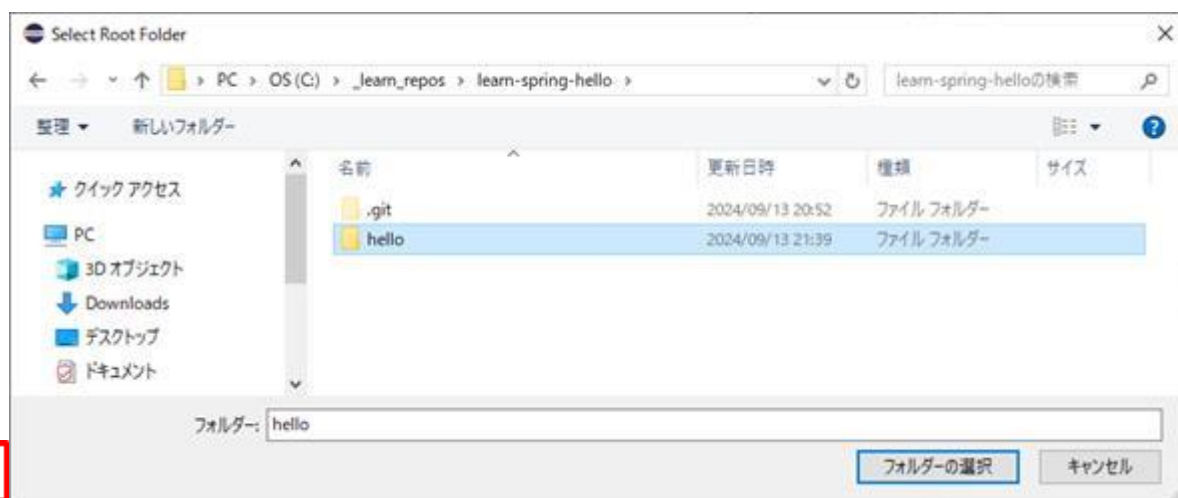
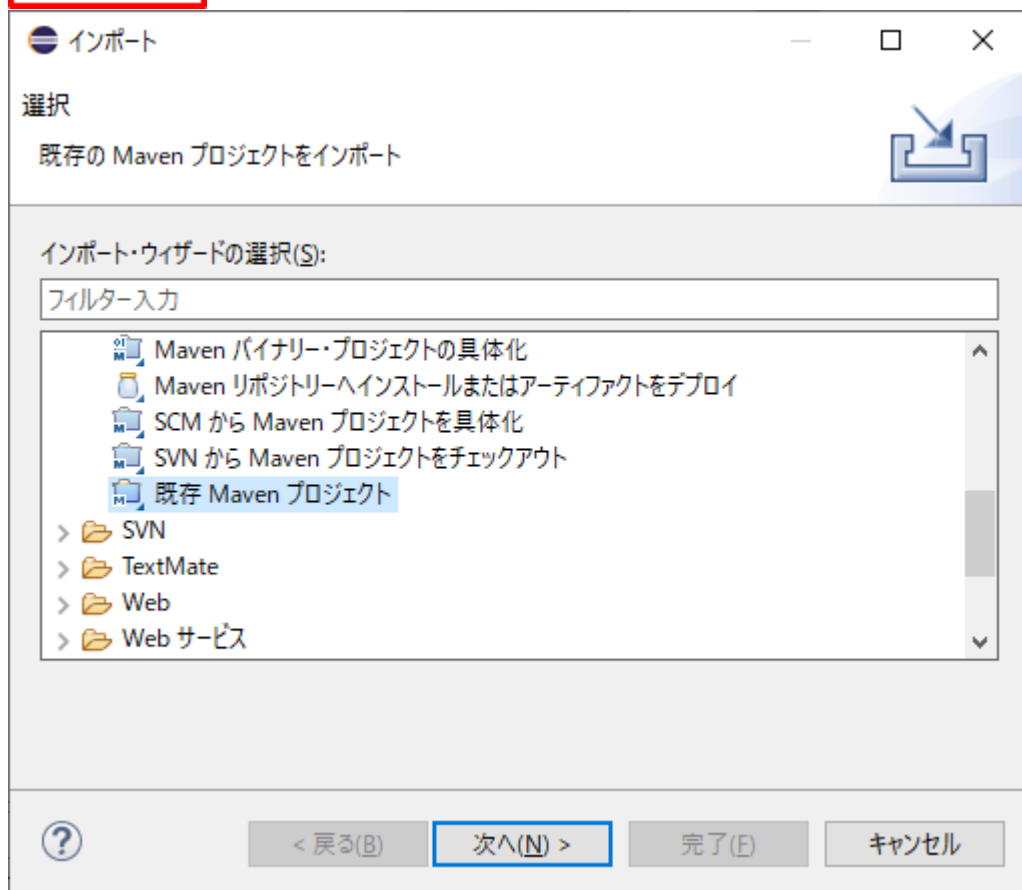
- Spring Web
- Spring Data JPA
- PostgreSQL Driver
- H2 Database
- Thymeleaf
- Spring Boot DevTool

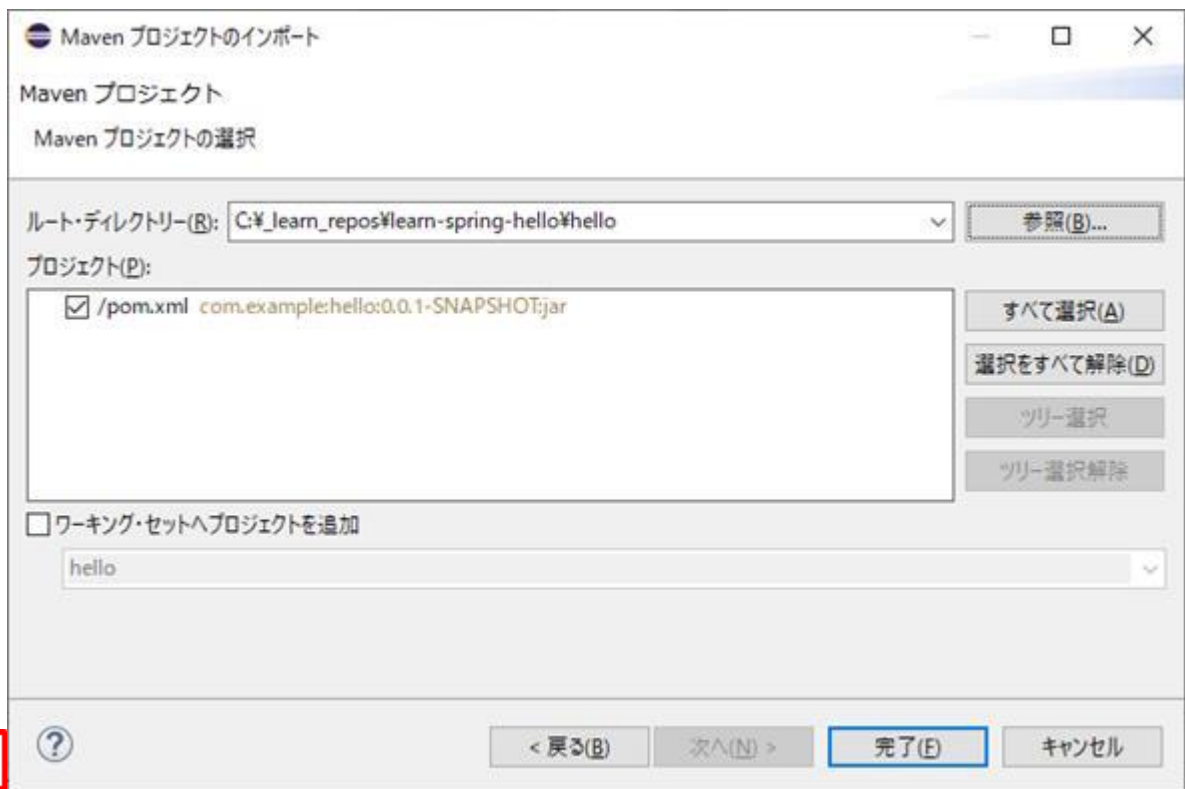



```
| .gitignore
| HELP.md
| mvnw
| mvnw.cmd
| pom.xml
|
|— .mvn
|   |— wrapper
|       maven-wrapper.properties
|
|— src
|   |— main
|       |— java
|           |— com
|               |— example
|                   |— hello
|                       HelloApplication.java
|
|       |— resources
|           application.properties
|
|       |— static
|       |— templates
|
|— test
|   |— java
|       |— com
|           |— example
|               |— hello
|                   HelloApplicationTests.java
```

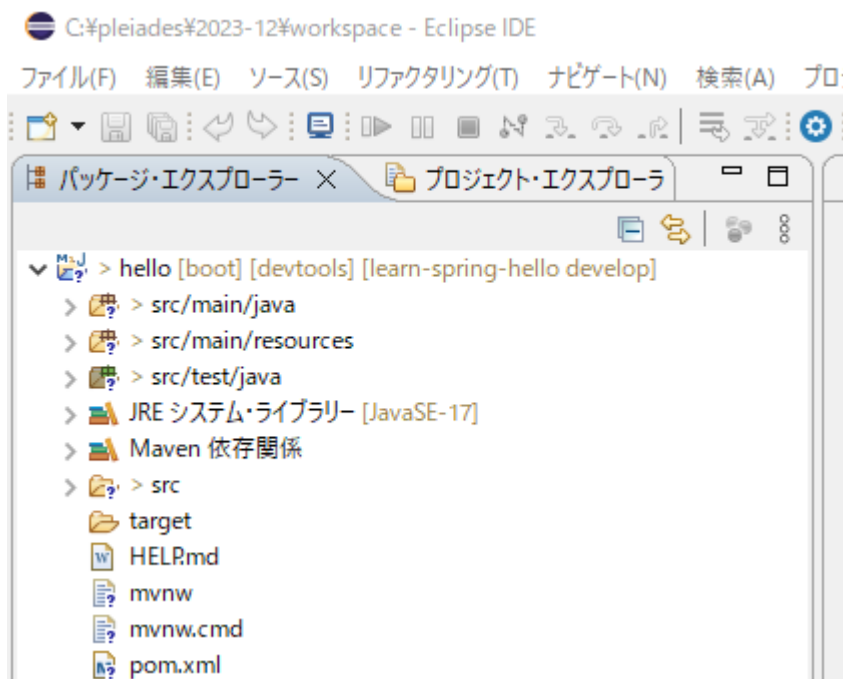
4.3 （手順3） 初期プロジェクトを Eclipse にインポート

Eclipse で、ファイル → インポート → 既存 Maven プロジェクト →





完了



4.4 （手順4）初期プロジェクトをカスタマイズ

初期プロジェクトのままでは HelloWorld ページは表示されませんので、次の4つのファイルを追加します。

HelloController.java

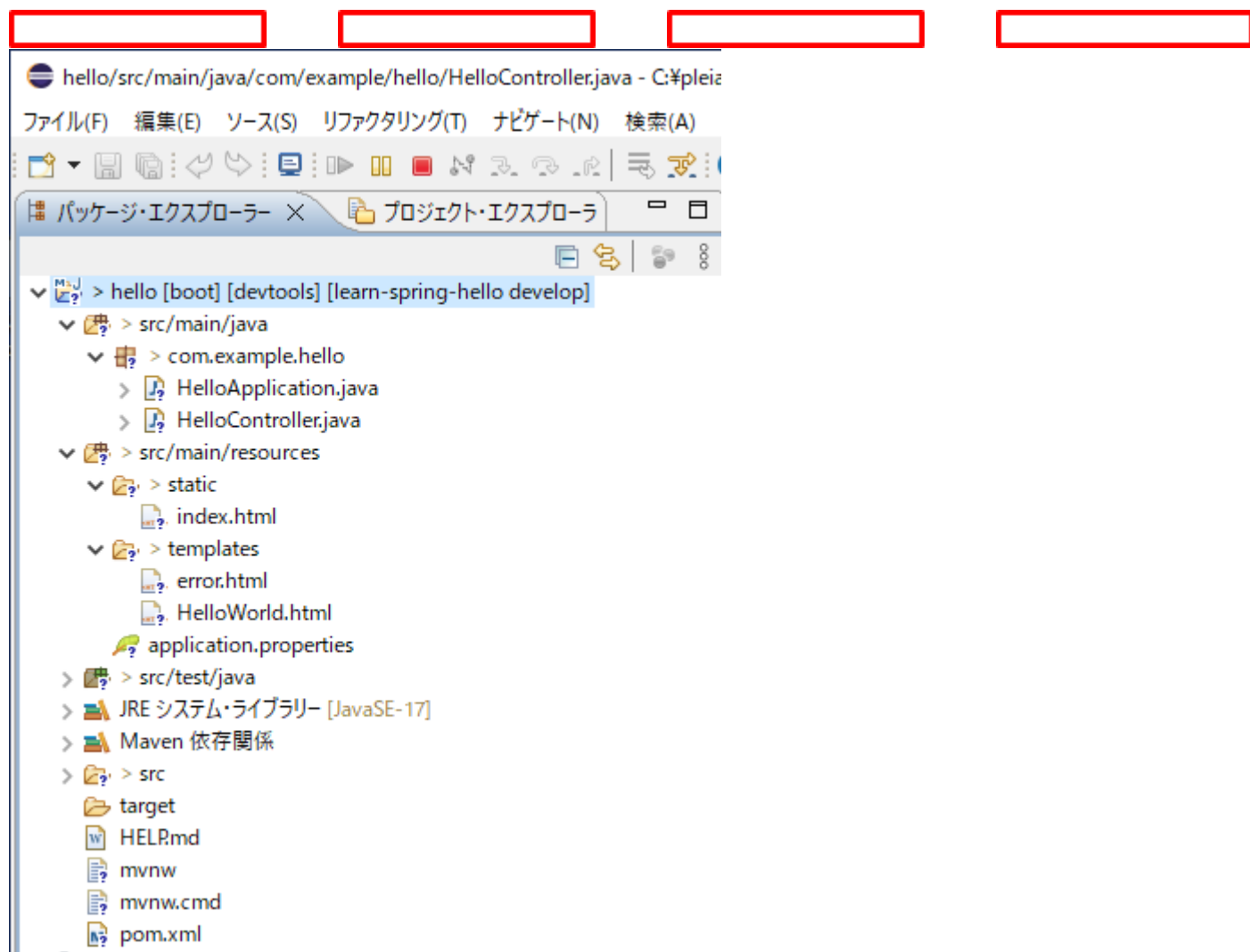
HelloWorld.html

error.html

index.html

追加する場所は次図の通りです。

4.4.1 カスタマイズの場所



4.4.2 カスタマイズの内容

(1) HelloController.java

```
package com.example.hello;
```

```
import org.springframework.stereotype.Controller;
```

```
import org.springframework.web.bind.annotation.GetMapping;
```

```
import org.springframework.web.servlet.ModelAndView;
```

```
@Controller
```



```
public class HelloController {  
  
    @GetMapping("hello")  
    public ModelAndView hello(ModelAndView mav) {  
        mav.setViewName("HelloWorld");  
        return mav;  
    }  
}
```

(2) HelloWorld.html

```
<!DOCTYPE html>  
<html>  
<head>  
<meta charset="UTF-8">  
<title>Insert title here</title>  
</head>  
<body>  
<h1>HelloWorld !!</h1>  
</body>  
</html>
```

(3) error.html

```
<!DOCTYPE html>  
<html>  
<head>  
<meta charset="UTF-8">  
<title>Insert title here</title>  
</head>  
<body>  
<h1>エラーページ</h1>  
</body>  
</html>
```

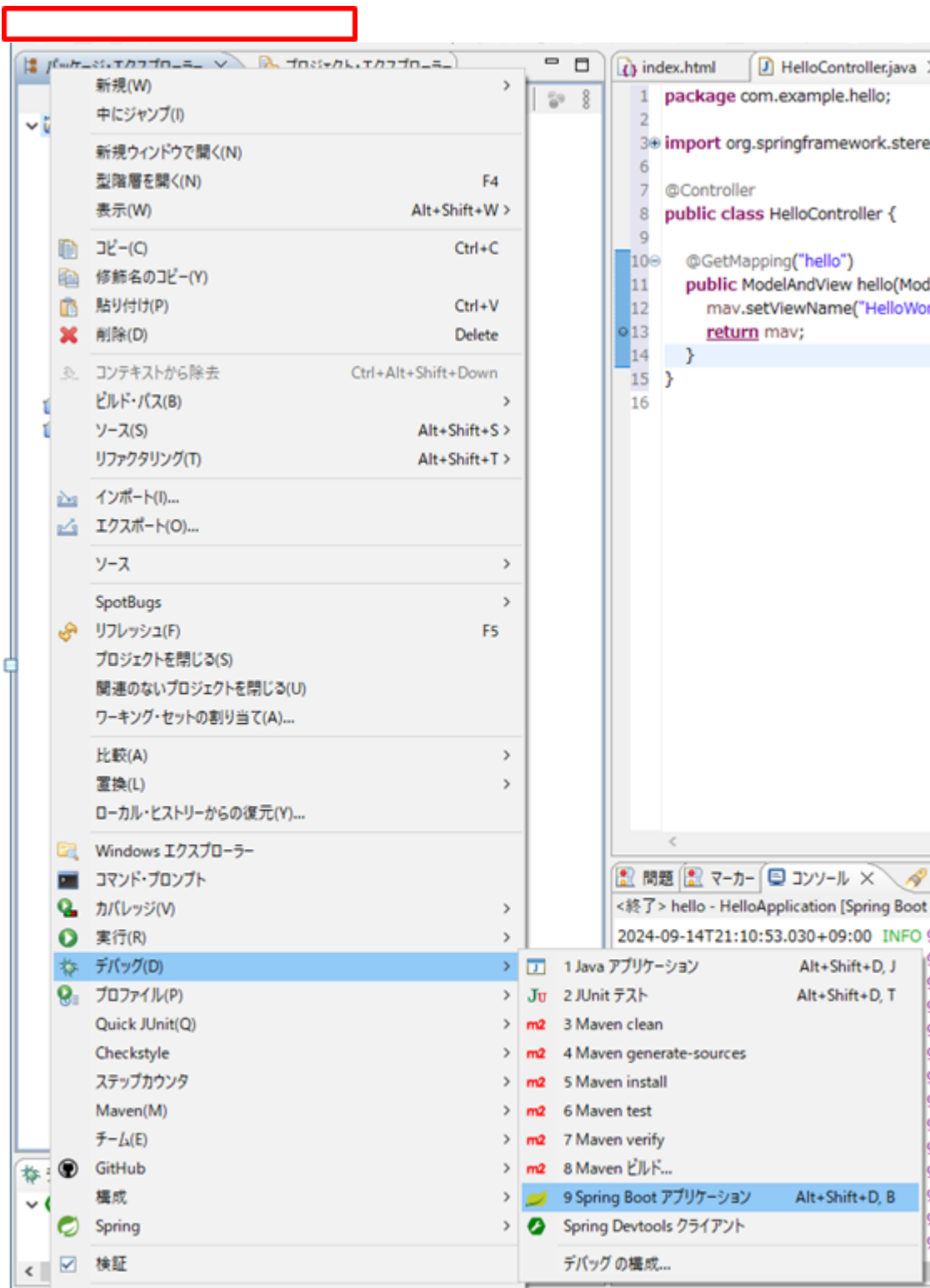
(4) index.html

```
<!DOCTYPE html>  
<html>  
<head>
```

```
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
<h1>静的 index.html ページ</h1>
</body>
</html>
```

4.5 Web アプリケーションを起動

Eclipse のパッケージエクスプローラまたはプロジェクトエクスプローラで hello を右クリックします。
次図のように



Eclipse のコンソールに次のようなログが表示され、Web アプリケーションが起動したことを確認できます。

4.6 結果を確認

ブラウザを開いて次の URL にアクセスします。

<http://localhost:8080/>

静的index.htmlページ

<http://localhost:8080/hello>

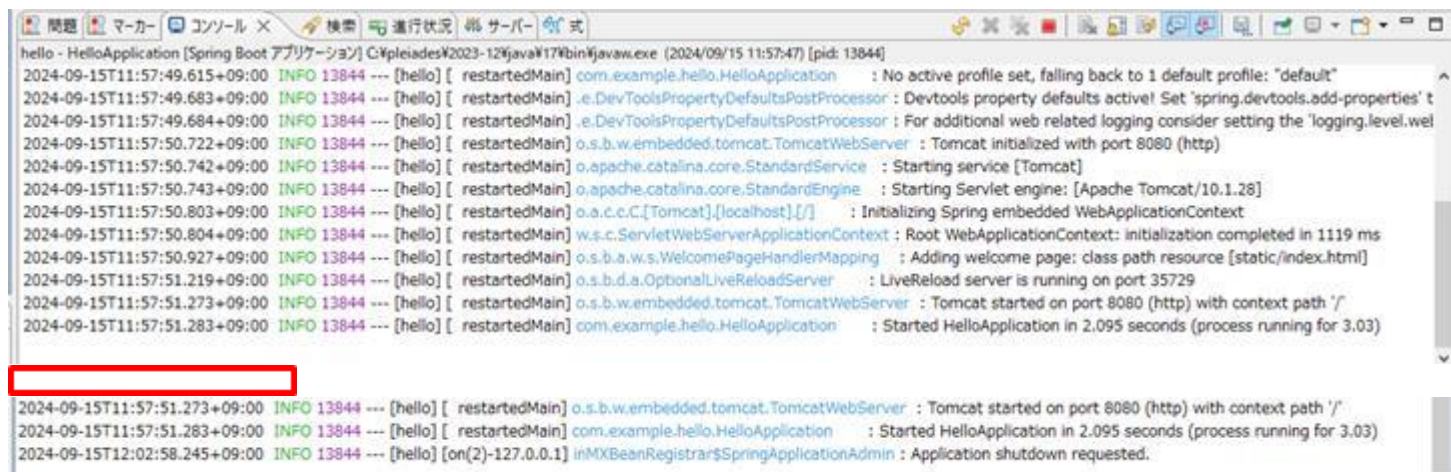
HelloWorld !!

<http://localhost:8080/hello999>

エラーページ

4.7 Web アプリケーションを停止

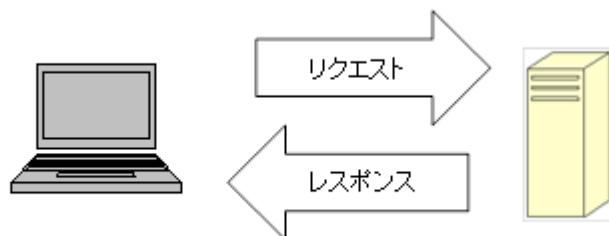
☐いくつか方法があります。一つは次図の赤いボタン押下で停止します。



```
hello - HelloApplication [Spring Boot アプリケーション] C:\pleiades\2023-12\java\17\bin\java.exe (2024/09/15 11:57:47) [pid: 13844]
2024-09-15T11:57:49.615+09:00 INFO 13844 --- [hello] [ restartedMain] com.example.hello.HelloApplication : No active profile set, falling back to 1 default profile: "default"
2024-09-15T11:57:49.683+09:00 INFO 13844 --- [hello] [ restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : Devtools property defaults active! Set 'spring.devtools.add-properties' to 'true' to enable adding custom properties
2024-09-15T11:57:49.684+09:00 INFO 13844 --- [hello] [ restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : For additional web related logging consider setting the 'logging.level.web' property to 'TRACE'
2024-09-15T11:57:50.722+09:00 INFO 13844 --- [hello] [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port 8080 (http)
2024-09-15T11:57:50.742+09:00 INFO 13844 --- [hello] [ restartedMain] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2024-09-15T11:57:50.743+09:00 INFO 13844 --- [hello] [ restartedMain] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.28]
2024-09-15T11:57:50.803+09:00 INFO 13844 --- [hello] [ restartedMain] o.s.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2024-09-15T11:57:50.804+09:00 INFO 13844 --- [hello] [ restartedMain] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 1119 ms
2024-09-15T11:57:50.927+09:00 INFO 13844 --- [hello] [ restartedMain] o.s.b.a.w.s.WelcomePageHandlerMapping : Adding welcome page: class path resource [static/index.html]
2024-09-15T11:57:51.219+09:00 INFO 13844 --- [hello] [ restartedMain] o.s.b.d.a.OptionalLiveReloadServer : LiveReload server is running on port 35729
2024-09-15T11:57:51.273+09:00 INFO 13844 --- [hello] [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path '/'
2024-09-15T11:57:51.283+09:00 INFO 13844 --- [hello] [ restartedMain] com.example.hello.HelloApplication : Started HelloApplication in 2.095 seconds (process running for 3.03)
2024-09-15T11:57:51.273+09:00 INFO 13844 --- [hello] [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path '/'
2024-09-15T11:57:51.283+09:00 INFO 13844 --- [hello] [ restartedMain] com.example.hello.HelloApplication : Started HelloApplication in 2.095 seconds (process running for 3.03)
2024-09-15T12:02:58.245+09:00 INFO 13844 --- [hello] [on(2)-127.0.0.1] inMXBeanRegistrar$SpringApplicationAdmin : Application shutdown requested.
```

5. Web アプリケーションの設計と構造

前章の「簡単な Web アプリケーション」(Spring + Thymeleaf) では、最小限のリクエストとレスポンスを確認できました。しかし、ここから先を、自己流だけで機能を追加していても、実用的な品質のアプリケーションを作成することは困難でしょう。通常は、すでに多くの開発者によって使用されている設計手法や構造パターンを参考にして設計や構造を決めていきます。



次の一文は、エリックエヴァンスのドメイン駆動設計 日本語版第 4 章から引用したものです。

～引用ここから～

輸送アプリケーションで、貨物の荷出し地を都市の一覧から選択するという、ユーザの簡単な操作をサポートしたい。そのためには、次の処理を実行するプログラムコードがなければならない。①スクリーンにウィジェットを描き、②選択できる都市をすべてデータベースに問い合わせ、③ユーザの入力を解釈して検証し、④選択された都市を貨物と結び付け、⑤データベースへの変更をコミットする。このコードすべてが同じプログラムの一部になるが、輸送業というビジネスに関係するのは、そのごく一部にすぎない。

ソフトウェアプログラムには、さまざまな種類のタスクを実行する設計とコードがある。

ユーザ入力を受付け、ビジネスロジックを実行し、データベースにアクセスして、ネットワークを通じて通信し、ユーザに情報を表示するといった具合である。したがって、各プログラム機能に含まれるコードは、かなりの量になる可能性がある。

～引用ここまで～

単純なプログラムですが、十分な分析や設計をしないで、HelloWorld の延長のようなつもりで作成すると複雑で混沌とした Web アプリケーションになるでしょう。当然、拡張性や保守性の低いシステムとなります。そのためにはアプリケーションアーキテクチャというものを理解し適切に使用していく必要があります。

以下の用語は、すべて Web アプリケーションの設計や構造に関するものであり、互いに関連しています。それぞれが異なる視点からアプリケーションの構造を示していますが、目的は一貫して、コードの可読性や保守性、拡張性を高めることです。

5.1 アプリケーションアーキテクチャ

アプリケーションアーキテクチャとは、ソフトウェア全体の構造を定義し、どのように機能が分割されているか、各コンポーネントがどのように相互作用するかを示します。Web アプリケーションでは、複雑な要件を効率よく満たすために、設計時に適切なアーキテクチャが選ばれます。以下に説明する MVC、レイヤードアーキテクチャ、クリーンアーキテクチャはすべて、異なる種類のアプリケーションアーキテクチャです。

5.2 MVC (Model View Controller)

MVC は、Web アプリケーションの構造を「Model (データの管理)」「View (ユーザーインターフェース)」「Controller (ロジックと指令)」の 3 つに分ける設計パターンです。

- Model: データやビジネスロジックを担当します。データベースの操作やデータの整形などを行います。
- View: ユーザーに表示する部分です。HTML やテンプレートエンジンなどで、ユーザーインターフェースを生成します。
- Controller: ユーザーの入力を受け取り、Model と View を制御します。ユーザーからのリクエストを処理し、適切なレスポンスを返す役割を担います。

5.3 レイヤードアーキテクチャ

レイヤードアーキテクチャは、アプリケーションを層 (レイヤー) ごとに分割する設計です。一般的には、以下のような層に分けられます:

- プレゼンテーション層: ユーザーインターフェースを扱います (HTML、CSS、JavaScript など)。
- アプリケーション層: ビジネスロジックを実行し、アプリケーションの操作を管理します。
- ドメイン層: ビジネスルールやデータの処理を担当します。
- データアクセス層: データベースとのやり取りを行います。

このように層を分けることで、責任の分離が明確になり、保守や拡張がしやすくなります。

5.4 クリーンアーキテクチャ

クリーンアーキテクチャは、アプリケーションを柔軟かつ保守しやすくすることを目指した設計思想です。ヘキサゴナルアーキテクチャ、オニオンアーキテクチャなどの出現を受けてこれらの概念を統合しようとしたものです。ソフトウェアの依存関係が一方方向 (内側から外側) になるように設計します。基本的な構造は以下の通りです:

- エンティティ: ビジネスルールやデータモデルを表現します。
- ユースケース: システムがユーザーに提供するサービスや機能です。
- インターフェース (外側の層): プレゼンテーション層やデータベース層といった、システム外部とやり取りする部分です。

クリーンアーキテクチャの特徴は、外部依存 (データベース、UI、フレームワークなど) からビジネスロジックを分離することにより、変更に強い設計を実現する点です。

重要な関連概念

- SOLID 原則: オブジェクト指向設計の基本的な 5 つの原則で、保守性や拡張性を高めるためのガイドラインです。
- 依存性の逆転: クリーンアーキテクチャで重要な概念であり、具体的な実装がビジネスロジックに依存するのではなく、ビジネスロジックがインターフェースに依存するという考え方です。
- 依存性注入 (DI) : アプリケーションの依存関係を外部から注入することで、モジュール間の結合を緩める設計手法です。Spring フレームワークなどでよく使われます。

これらの概念は、Web アプリケーション開発におけるコードの可読性、テストのしやすさ、保守性を向上させるために非常に役立ちます。

6. Java パッケージ構成の例

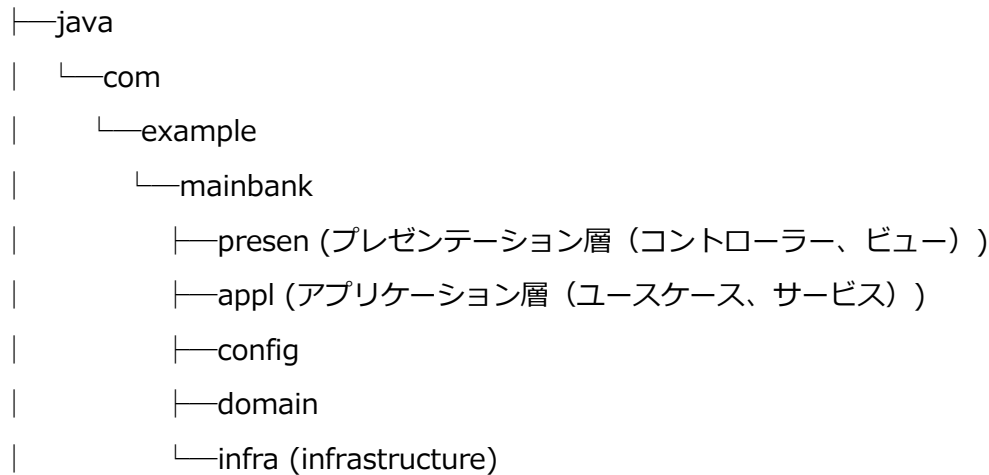
本章の内容は例です。

6.1 層型アーキテクチャ

層毎に役割を明確に分けます。トランザクション・スクリプトと呼ばれるアンチパターンとならないように、そして、低品質で保守や拡張が困難な混沌としたソフトウェアとならないように、適切に役割を分けます。各層の名前や構成はチームで決めていきます。必要な場合はリファクタリングします。

| 名前 | 役割 |
|------------------------|---|
| プレゼンテーション層 (UI 層) | アクター（人間や別のコンピュータ）と情報を相互通信します。 |
| アプリケーション層 (ユースケース層) | ユースケース、シナリオを実行します。 |
| ドメイン層 | ドメイン駆動設計などでモデリングしたクラス群によってドメインロジックを提供します。 |
| インフラストラクチャ層 | 上位のレイヤを支える一般的な技術機能（永続化機能やデータ送受信機能など）を提供します。 |

6.2 レイヤードアーキテクチャ的な例



```
└resources
  └static
    └css
  └templates
```

6.3 クリーンアーキテクチャ的な例

クリーンアーキテクチャでは、依存関係の方向を意識し、ビジネスロジックを中心に据えるような構成が推奨されます。

```
└java
  └com
    └example
      └mainbank
        └config
        └domain
          └model
            └repository (repository interface)
          └application
            └usecase (business logic)
          └infra (implementation of repository, external services, etc.)
          └adapter (controllers, gateways, and presenters)
          └shared (common utilities)
└resources
  └static
  └templates
```

- domain: ビジネスルールに直接関係するクラス（エンティティ、バリューオブジェクト、リポジトリインターフェース）を配置します。
- application: ユースケースやビジネスロジックを表現するクラスを配置します。
- infra: 外部システムやデータベースに依存する実装をここに置きますが、domain に直接依存させない形にします。
- adapter: プレゼンテーション層や外部インターフェースに関する処理をまとめます。

7. 取引銀行管理システム（サンプルアプリ）

7.1 アプリケーションの要件

自社が取引する銀行の情報を管理する。

【銀行の種類】

- ・都市銀行
- ・地方銀行
- ・ネット銀行

【取引の種類】

- ・融資 (Loan)
- ・預金 (Deposit)
- ・送金

7.2 ユースケース

| | | |
|---|---------|---|
| 1 | ユースケース名 | 取引銀行を登録する。 |
| | アクター | 経理部員。 |
| | 事前条件 | 登録する取引銀行は未登録であること。 |
| | 事後条件 | エラーなしの場合は、登録済みとなること。 エラーありの場合は、未登録のままであること。 |
| | 主シナリオ | 1. ブラウザから所定の URL にアクセスし、取引銀行一覧画面を開く。 2. 新規登録ボタンを押下して新規登録画面を開く。 3. 取引銀行名、銀行種類、取引種類を入力する。 4. 保存ボタンを押下する。 |
| | 副シナリオ | 保存ボタン押下でエラーメッセージが表示された場合 |
| | | エラーを修正し、再度保存ボタンを押下する。 または、キャンセルボタン押下で終了する。 |

7.3 プロジェクト作成時の手順

サンプルアプリ作成時の Initializr の設定内容と GitHub で管理する手順について説明します。

7.3.1 Initializr で初期プロジェクトを作成

Project
☐ Gradle - Groovy
☐ Gradle - Kotlin
☒ Maven

Language
☒ Java
☐ Kotlin
☐ Groovy

Spring Boot
☐ 3.4.0 (SNAPSHOT) ☐ 3.4.0 (M3) ☐ 3.3.5 (SNAPSHOT)
☒ 3.3.4 ☐ 3.2.11 (SNAPSHOT) ☐ 3.2.10

Project Metadata

Group

Artifact

Name

Description

Package name

Packaging ☒ Jar ☐ War

Java ☐ 23 ☐ 21 ☒ 17

Dependencies ADD DEPENDENCIES... CTRL + B

Spring Web WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Data JPA SQL

Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

MySQL Driver SQL

MySQL JDBC driver.

H2 Database SQL

Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2mb) footprint. Supports embedded and server modes as well as a browser based console application.

Thymeleaf TEMPLATE ENGINES

A modern server-side Java template engine for both web and standalone environments. Allows HTML to be correctly displayed in browsers and as static prototypes.

Spring Boot DevTools DEVELOPER TOOLS

Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

GENERATE CTRL + G

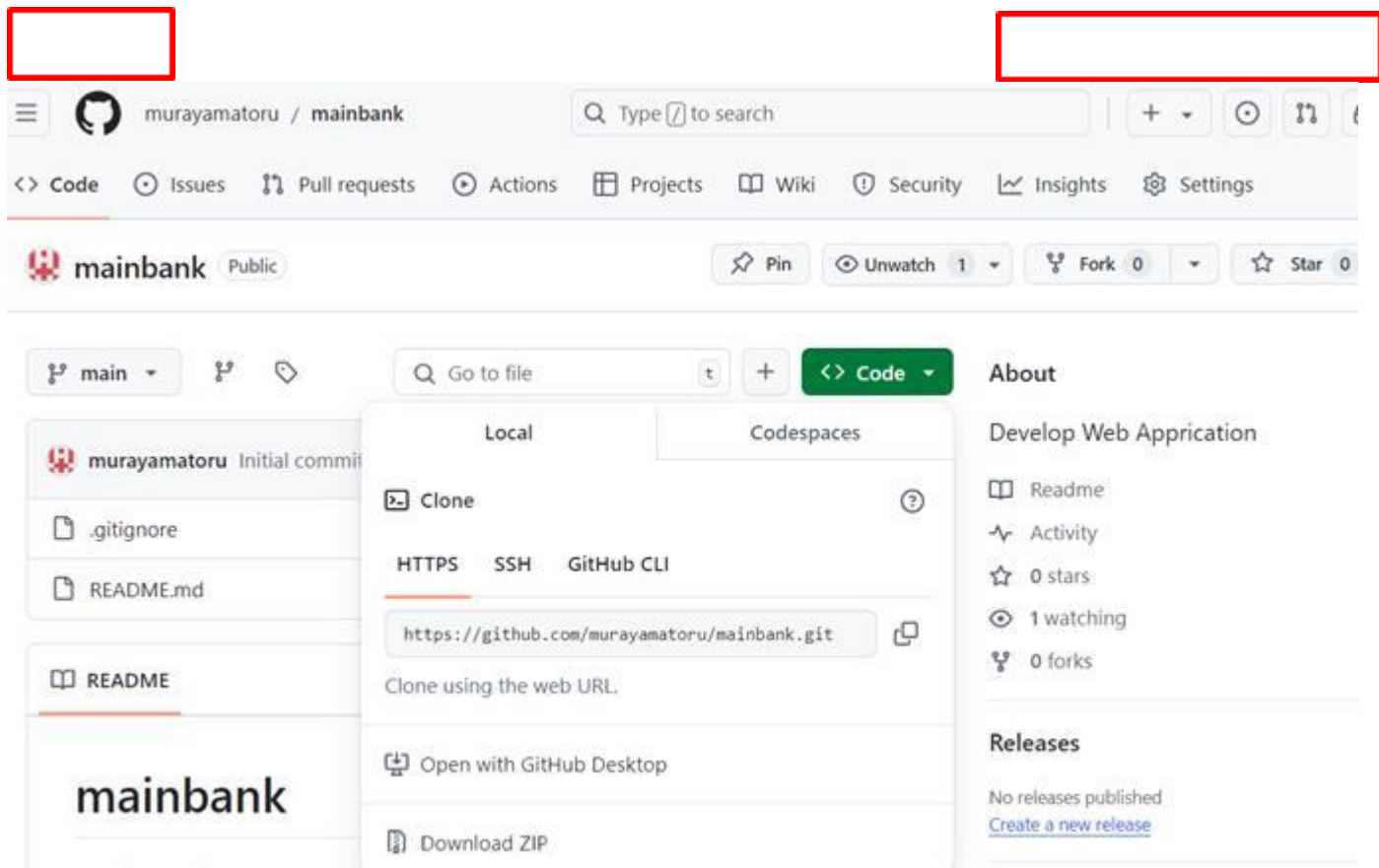
EXPLORE CTRL + SPACE

SHARE...

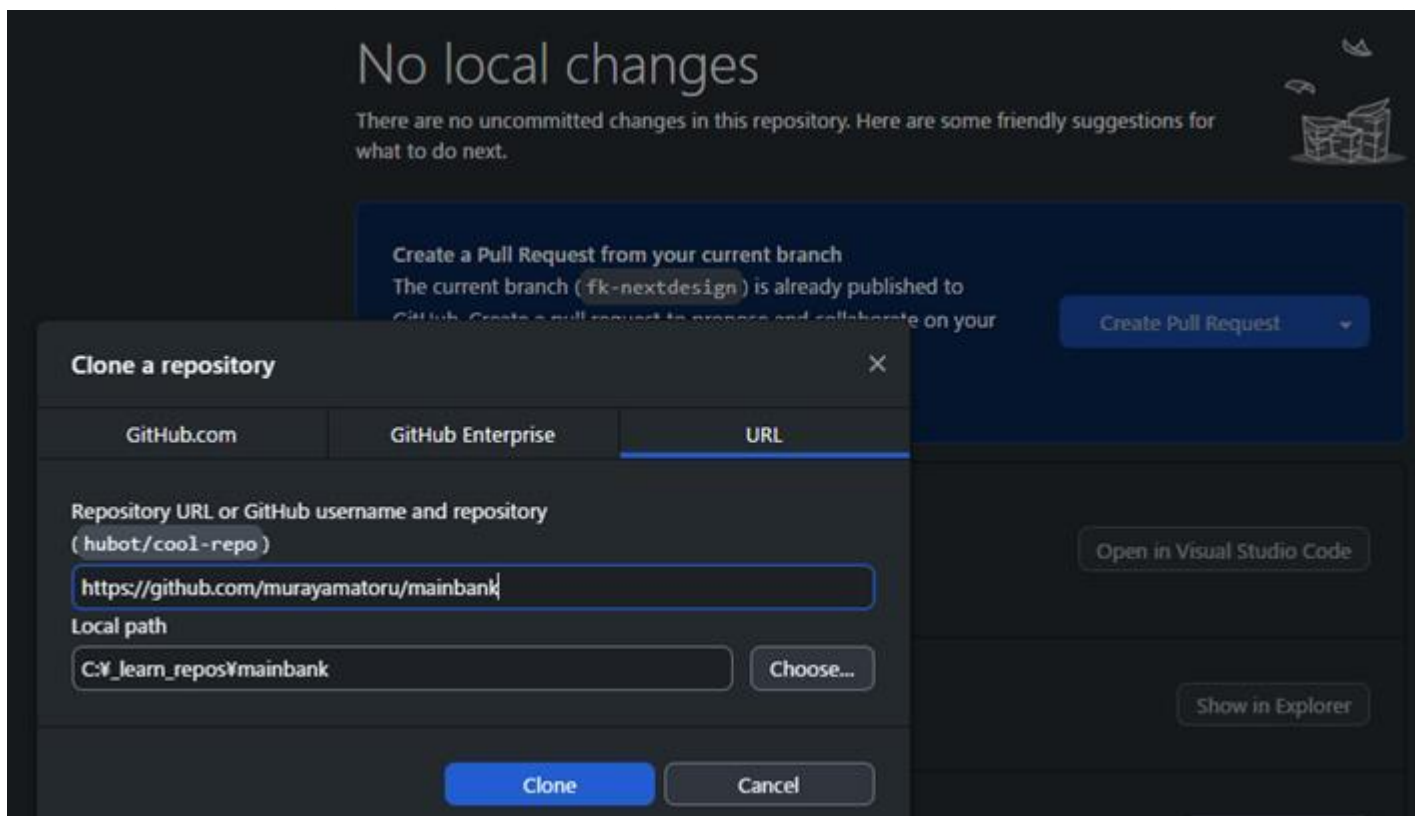
7.3.2 GitHub でリポジトリ mainbank を作成（任意：必要な方のみ）

これは GitHub でソースコードを管理したい場合のみ必要です。後から GitHub 管理もできます。

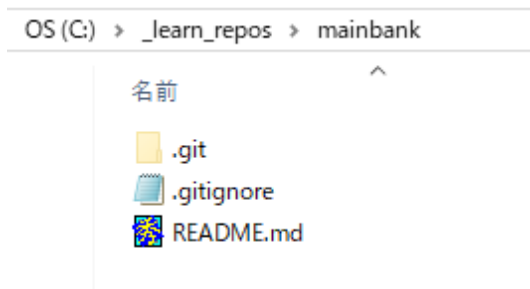
Code ボタンから「Open with GitHub Desktop」を選択すると自分の PC にインストール済みの GitHub Desktop が起動するので、GitHub Desktop を操作して、リポジトリをクローンします。



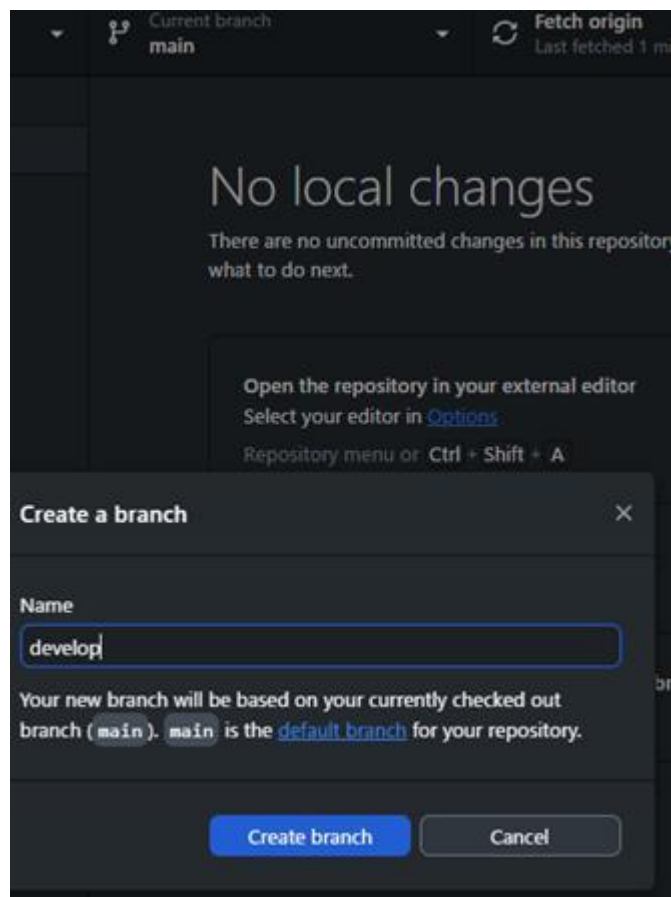
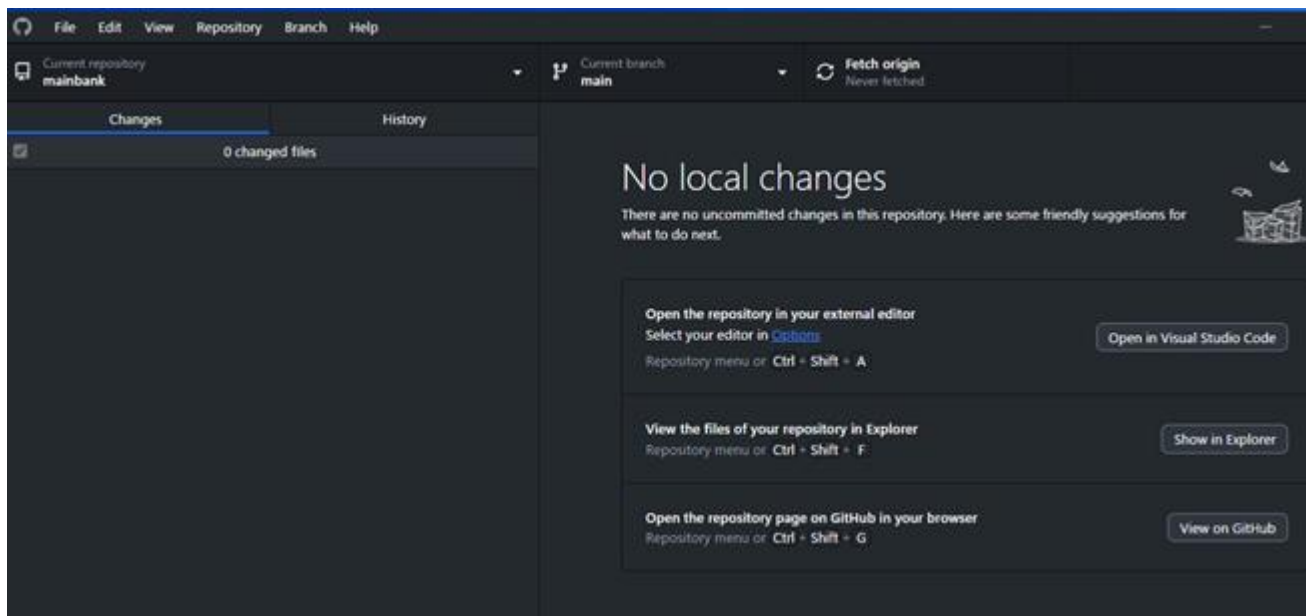
GitHub Desktop 画面が開きます



Clone すると Local Path のフォルダが自動生成されます。（そのつもりで指定してください）
次のような結果になります。

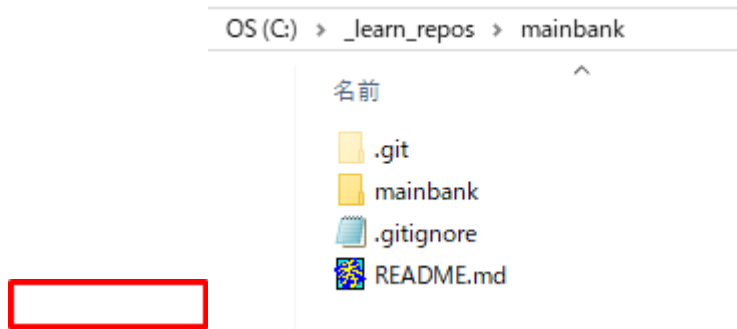


カレントブランチは、デフォルトの main ブランチになっています。

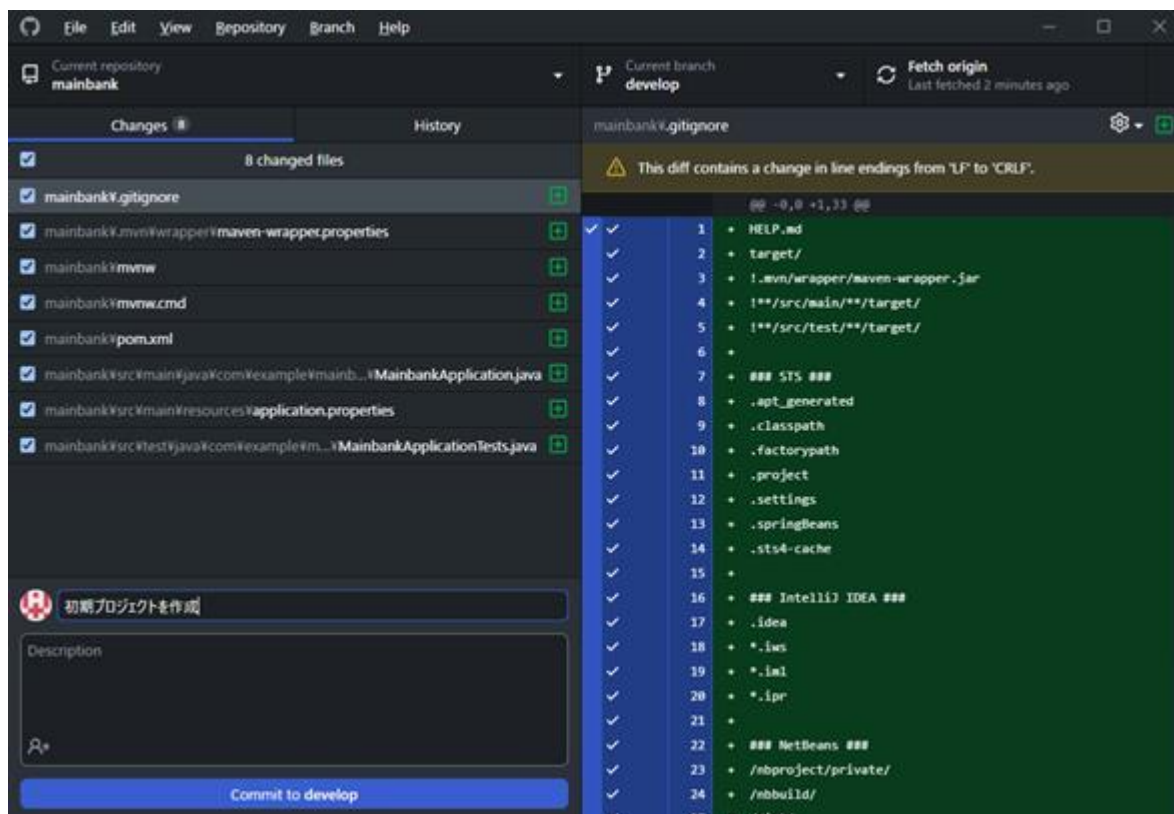


この例では、main ブランチから develop ブランチを作成し、develop をカレントブランチにしました。

ローカルフォルダに初期プロジェクトを解凍した後の mainbank を移動（コピー＆貼付け）します。
この辺の手順は色々なやり方あると思います。



GitHub Desktop の状態



Commit コメントを入力 → Commit to develop → Push origin → Create Pull Request
→ ブラウザで Github ページが開く

Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also compare across forks. [Learn more about diff comparisons here.](#)

base: main ← compare: develop ✓ **Able to merge.** These branches can be automatically merged.

Add a title

初期プロジェクトを作成

Add a description

Reviewers 1
No reviews

Assignees 1

下の方にスクロールして Create pull request を押下します

Markdown is supported Paste, drop, or click to add files

Create pull request

Remember, contributions to this repository should follow our [GitHub Community Guidelines](#).

少し下にスクロール



 Open 初期プロジェクトを作成 #1
murayamatoru wants to merge 1 commit into `main` from `develop` 



Require approval from specific reviewers before merging

[Rulesets](#) ensure specific people approve pull requests before they're merged.

Add rule



Continuous integration has not been set up

[GitHub Actions](#) and [several other apps](#) can be used to automatically catch bugs and enforce style.



This branch has no conflicts with the base branch

Merging can be performed automatically.

Merge pull request

You can also [open this in GitHub Desktop](#) or view [command line instructions](#).

Merge pull request を押下



 Open 初期プロジェクトを作成 #1
murayamatoru wants to merge 1 commit into `main` from `develop` 



Merge pull request #1 from murayamatoru/develop

初期プロジェクトを作成

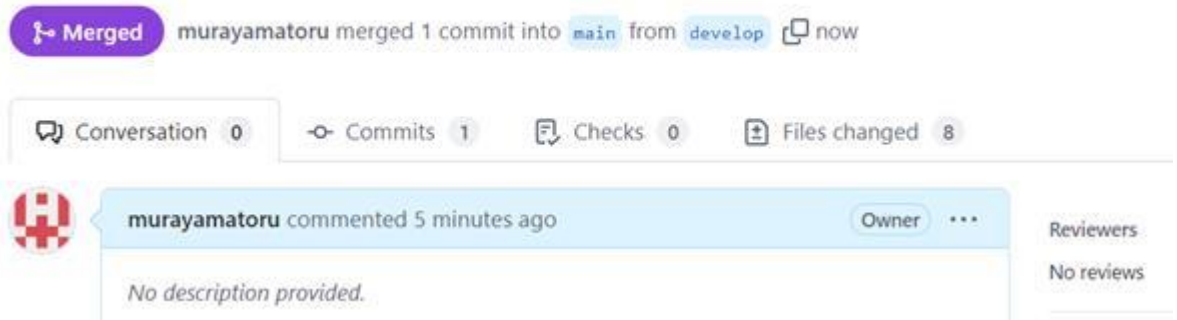
This commit will be authored by murayamatoru@users.noreply.github.com

Confirm merge

Cancel

Confirm merge を押下

初期プロジェクトを作成 #1



Merged が表示されたらマージ成功です。

Eclipse にインポートします。

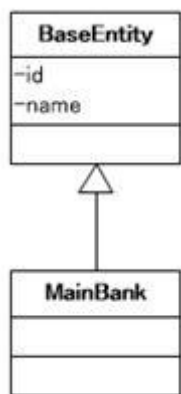
Eclipse ファイルメニュー → インポート → 既存 Maven プロジェクト → ルートディレクトリとして mainbank フォルダを選択 → 完了。

このインポートは mainbank を解凍し配置した後でも問題ありません。

7.4 ドメインモデルと永続化（継承の永続化戦略）

本書では Spring Boot JPA + MySQL を使用します。MySQL については後述します。また、取引銀行管理に登場する一部のドメインモデル名を使いますが、本セクションの目的は、継承関連を持つエンティティ（クラス）の永続化戦略です。そのため、エンティティのプロパティや振舞いはほぼ省略しています。

（クラス図）



（Java コード）

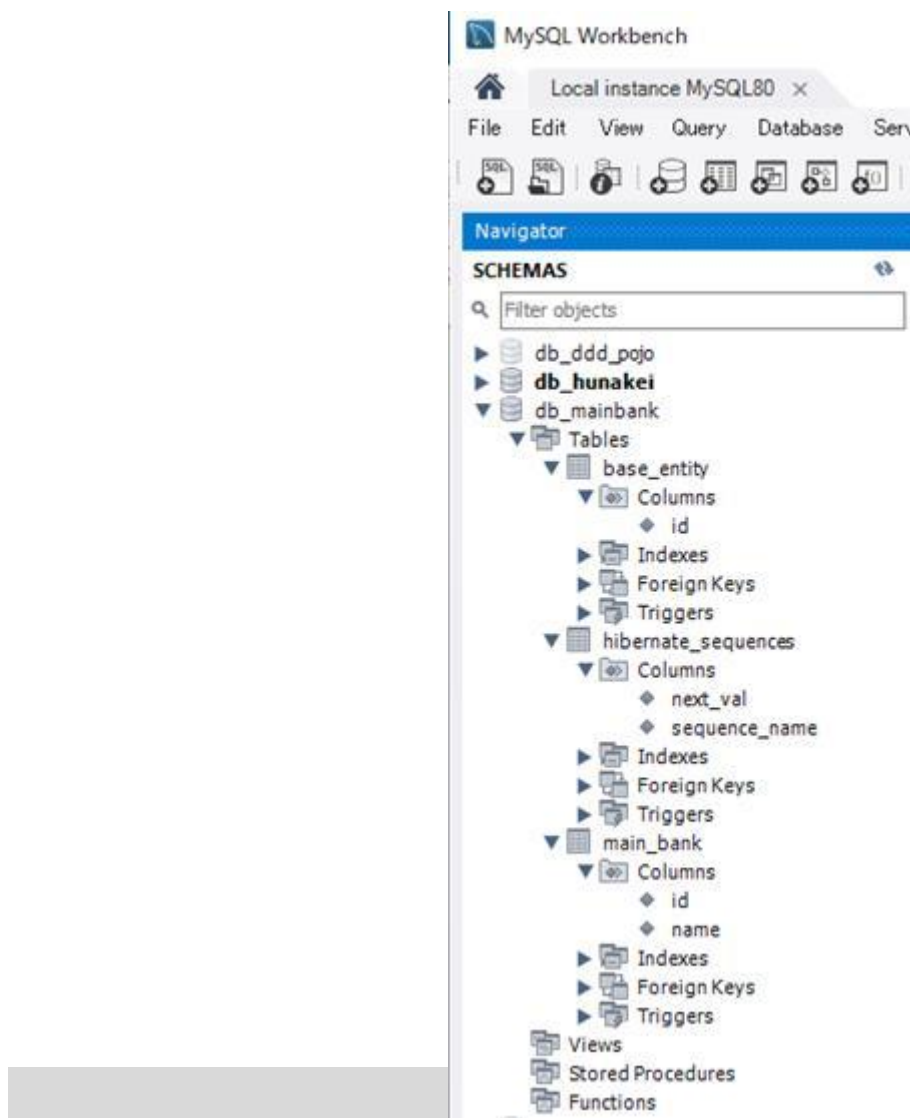
```
package com.example.mainbank.domain;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Inheritance;
import jakarta.persistence.InheritanceType;
```



```
/**
 * エンティの基底クラス
 */
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class BaseEntity {
    @Id
    //@GeneratedValue(strategy=GenerationType.IDENTITY)
    @GeneratedValue(strategy=GenerationType.TABLE)
    protected Long id;
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
}
```

```
package com.example.mainbank.domain;
import jakarta.persistence.Entity;
/**
 * 取引銀行
 */
@Entity
public class MainBank extends BaseEntity {
    private String name;
    //コンストラクタ
    public MainBank() {
        this.name = "";
    }
}
```

(MySQL データベーステーブルの状態)



(pom.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
```

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
```

```
https://maven.apache.org/xsd/maven-4.0.0.xsd">
```

```
<modelVersion>4.0.0</modelVersion>
```

```
<parent>
```

```
<groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter-parent</artifactId>
```

```
<version>3.3.4</version>
```

```
<relativePath/> <!-- lookup parent from repository -->
```

```
</parent>
```

```
<groupId>com.example</groupId>
```

```
<artifactId>mainbank</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>mainbank</name>
<description>Main Bank Management</description>
<url/>
<licenses>
  <license/>
</licenses>
<developers>
  <developer/>
</developers>
<scm>
  <connection/>
  <developerConnection/>
  <tag/>
  <url/>
</scm>
<properties>
  <java.version>17</java.version>
</properties>
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
```

```

        <scope>runtime</scope>
        <optional>true</optional>
    </dependency>
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>com.mysql</groupId>
        <artifactId>mysql-connector-j</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>

```

```

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
</project>

```

ここまで、クラス毎にテーブルを作成する戦略です。

ただし、@GeneratedValue(strategy = GenerationType.IDENTITY)と指定するとエラーになります。

原因は、InheritanceType.TABLE_PER_CLASS と GenerationType.IDENTITY の組み合わせにあります。

正しい指定方法は、@GeneratedValue(strategy = GenerationType.TABLE)です。

このように指定することで、Hibernate がテーブル (hibernate_sequences) を作成し、そこから ID を生成するので、TABLE_PER_CLASS と互換性があり、3つのテーブルが正しく作成されます。

この戦略は適切な解決策と言えます。ただし、ID テーブルが追加で作成されることに注意が必要です。

この他の戦略としては、

InheritanceType.JOINED や InheritanceType.SINGLE_TABLE があります。この戦略も検討可能です。

PostgreSQL の場合は、

PostgreSQL は IDENTITY 戦略と TABLE_PER_CLASS の組み合わせが MySQL と同様に問題を引き起こす可能性があります。

しかし、PostgreSQL のシーケンスは MySQL より柔軟なので、次の戦略を推奨します。

GenerationType.SEQUENCE : PostgreSQL はシーケンスに適しており、Hibernate がシーケンスを自動で管理します。この場合、データベース側での追加のシーケンス作成が不要となり、効率的です。

例 :

```
@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "your_sequence")
@SequenceGenerator(name = "your_sequence", sequenceName = "your_sequence_name",
allocationSize = 1)
```

まとめ

MySQL では、GenerationType.TABLE と TABLE_PER_CLASS の組み合わせが適切です。

PostgreSQL では、GenerationType.SEQUENCE の使用をお勧めします。

8. Spring 依存性注入とは

依存性注入 (DI : Dependency Injection) は、Spring フレームワークの中心的な機能で、オブジェクト同士の依存関係を効率的に管理する方法です。オブジェクトが他のオブジェクトに依存する際、その依存関係を外部から「注入」することで、コードを柔軟にします。

8.1 DI の基本的な考え方

通常、オブジェクトが別のオブジェクトを必要とする場合、そのオブジェクトを自分で作成します。しかし、これでは依存関係が強くなり、コードの再利用やテストが難しくなります。DI を使うと、必要な依存オブジェクトを外部（コンテナ）から提供されるため、オブジェクト間の結びつきが緩やかになります。

8.2 DI の例

例えば、以下のようなクラスがあるとします。

```
public class Service {
    private Repository repository;
    public Service(Repository repository) {
```

```

        this.repository = repository;
    }
    public void performTask() {
        repository.save();
    }
}

```

Service クラスは Repository クラスに依存しています。通常は Repository のインスタンスを Service クラス内で生成しますが、Spring では Service のコンストラクタに Repository を渡すことで依存性を外部から注入します。

例

```
this.repository = new ConRepository();
```

とすると、Service クラスは、Repository インタフェースを実装した具象クラス ConRepository と強く結びついてしまいます。

8.3 DI の仕組み

Spring では、DI を使ってオブジェクトの依存関係を設定できます。主な注入方法は 3 つあります。

- コンストラクティンジェクション

コンストラクタを通じて依存オブジェクトを注入する方法です。上記の例がこれに該当します。

- セッターインジェクション

セッターメソッドを使って依存オブジェクトを注入します。

```

public class Service {
    private Repository repository;
    public void setRepository(Repository repository) {
        this.repository = repository;
    }
}

```

- フィールドインジェクション

フィールドに直接注入する方法で、@Autowired アノテーションを使います。

```

public class Service {
    @Autowired
    private Repository repository;
}

```

8.4 DI のメリット

- テストがしやすい: 依存関係を外部から注入するため、テスト時にモックオブジェクトを簡単に差し替えることができます。
- コードの再利用性が高まる: 依存オブジェクトを外部から渡すことで、同じクラスを異なるシチュエーションで再利用できます。
- 結合度が低くなる: クラス間の依存関係を疎結合に保つことができ、コードが柔軟で変更に強くなります。

依存性注入は、より柔軟で保守性の高いアプリケーションを作るための非常に重要な概念です。Spring フレームワークはこの仕組みを自動的に管理してくれます。

8.5 依存性とは

「依存性」とは、何かが他の何かに頼って動く関係のことです。例えば、車がガソリンに「依存」しているとしましょう。車はガソリンがないと動かないので、ガソリンに頼っている、つまり依存しているというわけです。ソフトウェアでも、あるプログラム（車）が他のプログラム（ガソリン）を必要として動作する場合、その2つの間に「依存性」があるといえます。

8.6 注入とは

「注入」とは、必要なものを外から与えることです。車の例でいうと、車がガソリンスタンドで給油されるように、外からガソリンを「注入」されることです。つまり、車は自分でガソリンを作るのではなく、誰かがガソリンを入れてあげている状態です。

ソフトウェアの場合も同様で、必要なプログラム（例えば車でいうガソリン）を別のプログラムに「注入」して、うまく動くようにします。

依存性は、「あるものが他のものに頼っている関係」です。例えば、車がガソリンに頼って動いている状態。

注入は、「必要なものを外から与えること」です。車が自分でガソリンを用意するのではなく、外からガソリンを入れてもらイメージです。

このように考えると、依存性と注入の関係は、何かを動かすために必要なものを外から提供してもらう仕組みだと言えます。

9. Spring アノテーション

| 分類 | アノテーション | |
|----------------|--|---|
| 依存性注入 (DI) | @Autowired @Bean @Component @Configuration @Value @Resource @Inject @EJB @ManagedBean @Named | 依存性の注入や Bean の管理に使用するアノテーション。 |
| Web MVC | @Controller @RestController @RequestMapping @GetMapping @PostMapping @RequestParam @ModelAttribute @ResponseBody @ControllerAdvice | Web アプリケーションのコントローラやリクエストマッピングに使用。 |
| データアクセス (JPA) | @Entity @Table @Id @Repository @PersistenceContext | データベースとやり取りするエンティティやリポジトリに関するアノテーション。 |
| トランザクション管理 | @Transactional | トランザクションの管理やロールバックに関するアノテーション。 |
| アプリケーション設定・ブート | @SpringBootApplication @EnableScheduling | Spring Boot アプリケーションの設定や機能有効化に関するアノテーション。 |
| Lombok | @Data | コードの簡略化を目的とした Lombok ライブラリのアノテーション。 |

9.1 主要なアノテーション

(1) @Autowired

短い説明：Spring による依存性注入を行うアノテーション。

記述場所：フィールド、コンストラクタ、またはメソッド。

目的：Spring が管理する任意の Bean（オブジェクト）を注入するために使われます。

利用範囲：Spring の DI コンテナで管理されている全てのオブジェクトに適用できます。例えば、サービスクラスやリポジトリクラスなどに依存するオブジェクトを注入する際に使います。

例

@Autowired

```
private MyService myService;
```

@Autowired を使うことで、MyService クラスのインスタンスが自動的に注入され、myService というフィールドに格納されます。

補足：@Autowired は、デフォルトで by type（型に基づいて）自動的に適切な Bean を見つけて注入します。

必要に応じて@Qualifier を使い、特定の Bean を選択することも可能です。

(2) @Bean

短い説明：Spring コンテナに管理される Bean を定義するアノテーション。

記述場所：メソッド。

(3) @Component

短い説明：Spring コンテナに管理される一般的な Bean を定義するアノテーション。

記述場所：クラス。

(4) @Configuration

短い説明：Spring の設定クラスを示すアノテーション。

記述場所：クラス。

(5) @Controller

短い説明：Web リクエストを処理するコントローラを定義するアノテーション。

記述場所：クラス。

(6) @RestController

短い説明：REST API のエンドポイントを定義するコントローラ用のアノテーション

(@Controller + @ResponseBody)。

記述場所：クラス。

(7) @ResponseBody

短い説明：メソッドの戻り値を HTTP レスポンスのボディに直接変換するアノテーション。

記述場所：メソッド。

(8) @RequestMapping

短い説明：URL パスとリクエストハンドラをマッピングするアノテーション。

記述場所：クラスまたはメソッド。

(9) @GetMapping

短い説明：HTTP GET リクエストを処理するメソッドをマッピングするアノテーション。

記述場所：メソッド。

(10) @PostMapping

短い説明：HTTP POST リクエストを処理するメソッドをマッピングするアノテーション。

記述場所：メソッド。

(11) @RequestParam

短い説明：HTTP リクエストのパラメータをメソッドの引数にバインドするアノテーション。

記述場所：メソッド引数。

(12) @ModelAttribute

短い説明：モデルデータをリクエストと共にバインドするアノテーション。

記述場所：メソッド、またはメソッド引数。

(13) @ControllerAdvice

短い説明：全体的な例外処理やモデルデータの共通処理を提供するアノテーション。

記述場所：クラス。

(14) @Repository

短い説明：データアクセス層（DAO）の Bean を定義するアノテーション。

記述場所：クラス。

(15) @Entity

短い説明：JPA エンティティクラスを示すアノテーション。

記述場所：クラス。

(16) @Table

短い説明：エンティティがマッピングされるデータベーステーブルを指定するアノテーション。

記述場所：クラス。

(17) @Id

短い説明：エンティティの主キーを指定するアノテーション。

記述場所：フィールドまたはメソッド。

(18) @Data

短い説明：Lombok による自動的なゲッター、セッター、toString 等の生成アノテーション。

記述場所：クラス。

(19) @Service

短い説明：ビジネスロジックを持つサービスクラスを定義するアノテーション。

記述場所：クラス。

(20) @Transactional

短い説明：メソッドやクラスに対してトランザクションの境界を定義するアノテーション。

記述場所：クラスまたはメソッド。

(21) @SpringBootApplication

短い説明：Spring Boot アプリケーションのエントリーポイントを定義するアノテーション。

記述場所：クラス。

(22) @EnableScheduling

短い説明：スケジューリング機能を有効にするアノテーション。

記述場所：クラス。

(23) @PersistenceContext

@Autowired と @PersistenceContext はどちらも依存性注入を行うためのアノテーションですが、注入する対象や役割が異なります。

目的：特定の EntityManager を注入するために使われます。EntityManager は、データベース操作を行う際に必要な JPA (Java Persistence API) のインターフェースです。

利用範囲：主にデータベースにアクセスするための JPA (Java Persistence API) 関連クラスで使用されます。

@PersistenceContext を使うことで、Spring が管理する EntityManager を注入し、データベースとのやり取りを実現します。

例：

```
@PersistenceContext
```

```
private EntityManager entityManager;
```

@PersistenceContext を使うと、EntityManager のインスタンスが自動的に注入され、entityManager というフィールドで利用できるようになります。

補足: EntityManager は、エンティティ（データベーステーブルの行に対応するオブジェクト）を操作するために使用されます。例えば、エンティティの保存、更新、削除、クエリ実行などを行うために重要です。

注入対象:

@PersistenceContext は EntityManager の注入に特化しています。

用途:

@PersistenceContext は、JPA を使ったデータベース操作でのみ使います。

補足:

@PersistenceContext の代わりに@Autowired を使うと、次のような問題が発生する可能性があります。

- トランザクションの管理

@PersistenceContext は、JPA の EntityManager を注入する際、トランザクションスコープで管理されるため、トランザクション開始や終了のタイミングに応じて自動的に正しい EntityManager が提供されます。これにより、エンティティの状態が一貫性を保ち、トランザクションの境界内で適切に動作します。

一方、@Autowired は Spring の通常の Bean を注入するためのものです。もし @Autowired で EntityManager を注入しようとすると、Spring がトランザクションの境界を適切に管理できず、複数のトランザクション間で問題が発生する可能性があります。例えば、トランザクション内で EntityManager が正しく解放されない、または使い回されるといった不具合が起きます。

- スレッドセーフ性

@PersistenceContext を使うと、トランザクションスコープで EntityManager が注入されるため、マルチスレッド環境でもスレッドごとに適切に分離された EntityManager が使用されます。

@Autowired を使った場合、EntityManager がシングルトンとして扱われる可能性があり、スレッド間で共有されるとスレッドセーフでなくなることがあります。これにより、同時実行中の複数のトランザクションで不整合が発生する可能性があります。

そのため、EntityManager の注入には @PersistenceContext を使うべきで、@Autowired ではトランザクション管理やスレッドセーフ性の問題が起きる可能性があります。

(24) @Resource

目的: Java の依存性注入を行うための標準アノテーションで、主に名前による依存性注入に使われます。

特徴: Spring の @Autowired が型に基づいて注入するのに対し、@Resource は名前に基づいて依存性を解決します。

使用例:

```
@Resource(name="myBean")  
private MyBean myBean;
```

(25) @Inject

目的: Java の標準依存性注入アノテーション (JSR-330) 。Spring の@Autowired に非常に似ていますが、標準の DI アノテーションです。

特徴: @Autowired とほぼ同じ動作をしますが、Spring に特化していないため、他の DI フレームワークでも使えることが特徴です。

使用例:

@Inject

```
private MyService myService;
```

(26) @EJB

目的: Java EE (Jakarta EE) の環境で使われるアノテーションで、**Enterprise Java Beans (EJB) **を注入するために使用します。

特徴: EJB を使用する際、@Autowired ではなく @EJB を使って EJB コンポーネントを注入します。

使用例:

@EJB

```
private MyEJBService ejbService;
```

(27) @Value

目的: プロパティファイルなどから値を注入する際に使用します。

特徴: 値を直接コード内に埋め込むのではなく、設定ファイルから動的に読み込む際に使います。

使用例:

@Value("\${app.name}")

```
private String appName;
```

(28) @Transactional

目的: メソッドやクラスに対してトランザクション境界を定義するためのアノテーションです。

特徴: トランザクション管理を明示的に行う必要がある場合、@Transactional を使用してトランザクションの開始・終了を自動的に管理します。

使用例:

@Transactional

```
public void saveData() {
```

```
    // データ保存処理
```

```
}
```

(??? 何関係) ???)

(29) @Component

説明: Spring コンテナに管理される汎用的な Bean を定義します。

付与場所: 任意のクラス。具体的な役割がない場合に使用。

(30) @Controller

説明: プレゼンテーション層のコントローラとして使用されるクラスを指定します。

付与場所: コントローラクラス。

(31) @Service

説明: ビジネスロジックを提供するサービス層のクラスを指定します。

付与場所: サービスクラス。

(32) @Repository

説明: データアクセス層 (DAO) のクラスを指定します。データ操作時の例外を変換する機能も提供します。

付与場所: リポジトリクラス。

(33) @RestController

説明: RESTful な Web サービスを提供するコントローラを指定します (@Controller と @ResponseBody を組み合わせたもの)。

付与場所: REST API のエンドポイントを提供するクラス。

(34) @ControllerAdvice

説明: アプリ全体で共有する例外処理やデータバインディングの設定を行うクラスを指定します。

付与場所: グローバルな例外処理やアドバイスロジックを持つクラス。

(35) @ManagedBean

説明: JSF (JavaServer Faces) のマネージド Bean を定義します。

付与場所: JSF アプリケーションで使用される Bean クラス。

(36) @Named

説明: CDI (Context and Dependency Injection) で名前付き Bean を定義します。

付与場所: CDI コンテナで管理される Bean クラス。

9.2 バリデーション関係

(1) @Validated

説明: バリデーションを有効化します。メソッドレベルやクラスに付与して、引数やフィールドの検証を実施。

付与場所: サービスクラス、コントローラのメソッド。

使用例:

@Service

```
public class BankService {  
    public void createAccount(@Validated Account account) { ... }  
}
```

(2) @AssertTrue

説明: フィールド値が true であることを検証します。

付与場所: エンティティのフィールド。

使用例:

@AssertTrue

```
private boolean isActive;
```

(3) @AssertFalse

説明: フィールド値が false であることを検証します。

付与場所: エンティティのフィールド。

使用例:

@AssertFalse

```
private boolean isSuspended;
```

(4) @Null

説明: フィールド値が null であることを検証します。

付与場所: エンティティのフィールド。

使用例:

@Null

```
private String notSetYet;
```

(5) @NotNull

説明: フィールド値が null でないことを検証します。

付与場所: エンティティのフィールド。

使用例:

@NotNull

```
private String name;
```

(6) @NotBlank

説明: 空文字や空白だけでない文字列を検証します。

付与場所: エンティティのフィールド。

使用例:

@NotBlank

```
private String username;
```

(7) @Max(value)

説明: 数値が指定値以下であることを検証します。

付与場所: エンティティのフィールド。

使用例:

@Max(100)

```
private int age;
```

(8) @Min(value)

説明: 数値が指定値以上であることを検証します。

付与場所: エンティティのフィールド。

使用例:

@Min(18)

```
private int age;
```

(9) @DecimalMax(value, inclusive)

説明: 小数点含む数値が指定値以下であることを検証します。

付与場所: エンティティのフィールド。

使用例:

@DecimalMax(value = "100.00", inclusive = false)

```
private BigDecimal discount;
```

(10) @DecimalMin(value, inclusive)

説明: 小数点含む数値が指定値以上であることを検証します。

付与場所: エンティティのフィールド。

使用例:

@DecimalMin(value = "0.01", inclusive = true)

```
private BigDecimal price;
```

(11) @Positive

説明: 正の数値であることを検証します。

付与場所: エンティティのフィールド。

使用例:

@Positive


```
private int quantity;
```

(12) @PositiveOrZero

説明: 0 または正の数値であることを検証します。

付与場所: エンティティのフィールド。

使用例:

```
@PositiveOrZero
```

```
private int stock;
```

(13) @Negative

説明: 負の数値であることを検証します。

付与場所: エンティティのフィールド。

使用例:

```
@Negative
```

```
private int balance;
```

(14) @NegativeOrZero

説明: 0 または負の数値であることを検証します。

付与場所: エンティティのフィールド。

使用例:

```
@NegativeOrZero
```

```
private int overdraft;
```

(15) @Digits(integer, fraction)

説明: 整数部と小数部の桁数を検証します。

付与場所: エンティティのフィールド。

使用例:

```
@Digits(integer = 6, fraction = 2)
```

```
private BigDecimal amount;
```

(16) @DateTimeFormat(pattern)

説明: 日時のフォーマットを指定します。

付与場所: エンティティのフィールド。

使用例:

```
@DateTimeFormat(pattern = "yyyy-MM-dd")
```

```
private LocalDate date;
```

(17) @Size(max, min)

説明: 配列やコレクション、文字列のサイズ（長さ）が指定範囲内であることを検証します。

付与場所: エンティティのフィールド。

使用例:

```
@Size(min = 5, max = 20)
```

```
private String password;
```

(18) @Email

説明: メールアドレス形式であることを検証します。

付与場所: エンティティのフィールド。

使用例:

```
@Email
```

```
private String emailAddress;
```

(19) @Pattern(regex, flags)

説明: 指定した正規表現に一致するか検証します。

付与場所: エンティティのフィールド。

使用例:

```
@Pattern(regexp = "^[a-zA-Z0-9]+$", flags = Pattern.Flag.UNICODE_CASE)
```

```
private String username;
```

9.3 JPA 関係

Spring Boot（JPA）でエンティティに関連を設定する際に使用する代表的なアノテーションについて、全体像とオプション、短い解説を以下にまとめます。これらのアノテーションは、データベース間のリレーション（関係）を表現するために用いられます。

(1) @OneToOne

概要: 1 対 1 のリレーションを表現します。たとえば、「1 つのユーザーが 1 つのプロファイルを持つ」場合などに使用します。

代表的なオプション:

- ・ mappedBy: 双方向関連時に使用。関連先のエンティティのフィールド名を指定。
- ・ cascade: カスケード操作（例: CascadeType.ALL）。
- ・ fetch: フェッチタイプ（例: FetchType.LAZY）。

```
@OneToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
```

```
@JoinColumn(name = "profile_id")
```

```
private Profile profile;
```

(2) @OneToMany

概要: 1 対多のリレーションを表現します。たとえば、「1 つの注文が複数の商品を含む」場合などに使用します。

代表的なオプション:

- mappedBy: 双方向関連時に使用。関連先のエンティティのフィールド名を指定。
- cascade: カスケード操作。
- fetch: フェッチタイプ。
- orphanRemoval: 親エンティティから削除された子エンティティを自動削除。

```
@OneToMany(mappedBy = "order", cascade = CascadeType.ALL, orphanRemoval = true)
```

```
private List<Item> items;
```

(3) @ManyToOne

概要: 多対 1 のリレーションを表現します。たとえば、「複数の商品が 1 つの注文に属する」場合などに使用します。

代表的なオプション:

- fetch: フェッチタイプ。
- optional: false の場合、関連先が必須となる。

```
@ManyToOne(fetch = FetchType.EAGER, optional = false)
```

```
@JoinColumn(name = "order_id")
```

```
private Order order;
```

(4) @ManyToMany

概要: 多対多のリレーションを表現します。たとえば、「複数の学生が複数のコースを受講する」場合などに使用します。

代表的なオプション:

- mappedBy: 双方向関連時に使用。
- cascade: カスケード操作。
- fetch: フェッチタイプ。
- 中間テーブルを明示する場合: @JoinTable。

```
@ManyToMany(cascade = CascadeType.ALL)
```

```
@JoinTable(
```

```
    name = "student_course",
```

```
    joinColumns = @JoinColumn(name = "student_id"),
```

```
    inverseJoinColumns = @JoinColumn(name = "course_id")
```

```
)
```

```
private Set<Course> courses;
```

オプションの解説

`cascade`: 親エンティティの操作(保存、削除など)を子エンティティに伝播させる。例: `CascadeType.PERSIST`, `CascadeType.REMOVE`, `CascadeType.ALL`。

`fetch`: リレーションのフェッチ戦略を指定。

- ・ `FetchType.LAZY`: 遅延ロード (必要になったときにデータを取得)。
- ・ `FetchType.EAGER`: 即時ロード (関連データをすぐ取得)。

`mappedBy`: 双方向関連の場合に、どちらが所有者かを指定。

`@JoinColumn`: 外部キー列を指定。

初心者向けポイント

関連を単方向に設計することで、シンプルに管理できます。

双方向関連を使う場合、必ず `mappedBy` を適切に設定してください。

`fetch` のデフォルト値に注意 (`@OneToMany` は `LAZY`、`@ManyToOne` は `EAGER`)。

さらに詳しく知りたい場合や具体的な設計に迷った際は、いつでもご質問ください！

JPA では、関連アノテーション以外にも、エンティティのフィールドに付与する重要なアノテーションがあります。以下に、代表的なものと簡単な解説をまとめます。

(5) `@Enumerated`

概要: `enum` 型をエンティティのフィールドとして使用する場合に付与します。

属性:

- ・ `EnumType.STRING` (推奨): `enum` の名前 (例: "ACTIVE", "INACTIVE") をデータベースに保存。
- ・ `EnumType.ORDINAL`: `enum` の順序番号 (例: 0, 1) を保存。ただし順序が変更されると問題が起きるため、非推奨。

```
public enum Status {  
    ACTIVE, INACTIVE  
}
```

```
@Enumerated(EnumType.STRING)
```

```
private Status status;
```

(6) `@Transient`

概要: データベースにマッピングしないフィールドに付与します。

たとえば、計算結果や一時的なデータを保持する場合に使用。

```
@Transient
```

```
private String temporaryValue;
```

(7) `@Lob`

概要: 大量のデータ (長い文字列やバイナリデータ) を保存するためのフィールドに付与します。

適用可能な型:

- ・ String: テキストデータ (CLOB: Character Large Object) 。
- ・ byte[] または Byte[]: バイナリデータ (BLOB: Binary Large Object) 。

@Lob

```
private String largeText;
```

(8) @Column

概要: テーブルのカラムのプロパティを設定します。

主な属性:

- ・ name: カラム名を指定。
- ・ nullable: true (デフォルト) でカラムの NULL 許可。
- ・ length: 文字列の最大長を指定 (デフォルト: 255) 。
- ・ unique: true でユニーク制約を設定。

```
@Column(name = "user_name", nullable = false, length = 100)
```

```
private String userName;
```

(9) @GeneratedValue

概要: 主キーの生成方法を指定します。通常、@Id と併用。

主な属性:

- ・ strategy: 主キーの生成戦略を指定。
 - ☐ GenerationType.IDENTITY: 自動インクリメント。
 - ☐ GenerationType.SEQUENCE: シーケンスを使用。
 - ☐ GenerationType.TABLE: テーブルを使用 (あまり一般的ではない) 。

@Id

```
@GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
private Long id;
```

(10) @Basic

概要: デフォルトで付与されているような動作を明示的に指定するアノテーション。

主な属性:

- ・ fetch: フェッチタイプを指定。
- ・ optional: true (デフォルト) で値が null を許可。

```
@Basic(optional = false)
```

```
private String email;
```

(11) @Temporal (JDK 1.8 以前推奨)

概要: Date または Calendar 型のフィールドに付与し、日付の保存形式を指定します。

属性:

- TemporalType.DATE: 日付のみ。
- TemporalType.TIME: 時刻のみ。
- TemporalType.TIMESTAMP: 日付と時刻。

@Temporal(TemporalType.TIMESTAMP)

private Date createDate;

注意: Java 8 以降は LocalDate や LocalDateTime を推奨。

初学者向けの把握ポイント

- @Enumerated: enum 型を使用する際に必須。
- @Transient: データベースに保存しないフィールドに必須。
- @Column と @GeneratedValue: 基本的なフィールド設定。
- @Lob: 大容量データを保存する場合。
- @Temporal (古いコード用) または Java 8 以降の LocalDateTime を活用。

10. コントローラとビューの間のデータの受け渡し

10.1 View → Controller

- @RequestParam

このアノテーションは、リクエストパラメータを引数として受け取る際に使用します。

主にフォームの入力や URL クエリパラメータを取得するのに使われます。

例えば、/greet?name=John のようなリクエストで「name」というパラメータを受け取る場合、@RequestParam を使います。

@GetMapping("/greet")

```
public String greet(@RequestParam("name") String name, Model model) {
    model.addAttribute("name", name);
    return "greet";
}
```

- @ModelAttribute

このアノテーションは、フォームデータやリクエストボディのデータをオブジェクトにバインドするために使用します。主にフォームのデータを複数のフィールドにまとめて受け取る場合に使われます。例えば、ユーザーオブジェクト全体をフォームから取得する場合に便利です。

```
@PostMapping("/register")
```

```
public String register(@ModelAttribute User user, Model model) {  
    model.addAttribute("user", user);  
    return "register";  
}
```

@RequestParam は個別のパラメータを、@ModelAttribute はオブジェクト全体を受け取るといった役割で使用されます。

他の方法

- @RequestBody

このアノテーションは、リクエストボディ全体を直接 Java オブジェクトに変換して受け取るのに使用します。特に、JSON 形式のデータを受け取る際に役立ちます。REST API でよく使います。

```
@PostMapping("/api/data")
```

```
public ResponseEntity<String> postData(@RequestBody Data data) {  
    // JSON をオブジェクトにバインド  
    return ResponseEntity.ok("Success");  
}
```

- PathVariable

パスの一部を引数として受け取る際に使います。

```
@GetMapping("/user/{id}")
```

```
public String getUser(@PathVariable("id") Long id, Model model) {  
    model.addAttribute("userId", id);  
    return "user";  
}
```

10.1.1 補足説明

Spring Boot では、フォームクラスに対応しない入力フィールドの値を受け取るためにさまざまな方法を提供しています。

たとえば、@RequestParam アノテーションを使用して直接リクエストパラメータを取得できます。

また、@ModelAttribute アノテーションを使用してフォームクラス以外のオブジェクトにデータをバインドできます。

要するに、Spring Boot ではフォームクラスを使用してフォームデータを受け取ることが一般的ですが、他の方法を使ってフォームクラスに定義されていない入力フィールドの値を受け取ることも可能です。

10.2 Controller → View

- Model

Controller から View にデータを渡す際に、Model を使ってデータを追加します。

これは最も一般的な方法です。

```
@GetMapping("/show")
public String show(Model model) {
    model.addAttribute("message", "Hello World");
    return "show";
}
```

- ModelAndView

ModelAndView は、モデルとビューを同時に返すことができるクラスです。

ビュー名とモデルを一緒に扱う場合に便利です。

```
@GetMapping("/show")
public ModelAndView show() {
    ModelAndView mav = new ModelAndView("show");
    mav.addObject("message", "Hello World");
    return mav;
}
```

- RedirectAttributes

リダイレクト時に一時的にデータを渡すために使用します。

RedirectAttributes は一度だけ使用される一時的なデータを渡すために便利です。

```
@PostMapping("/submit")
public String submit(RedirectAttributes redirectAttributes) {
    redirectAttributes.addFlashAttribute("message", "Data submitted successfully");
    return "redirect:/result";
}
```

これらの方法を組み合わせて、用途に応じたデータの受け渡しが可能です。

11. Spring アスペクト指向プログラミング

Spring AOP (Aspect-Oriented Programming) は、横断的な関心事 (cross-cutting concerns) をモジュール化するためのプログラミング手法です。横断的な関心事とは、アプリケーション全体にまたがる共通の機能のことを指し、ログ出力やトランザクション管理、セキュリティなどが典型例です。

AOP の基本的な考え方は、これらの横断的な機能をアプリケーションの本来のビジネスロジックとは分離し、個別に管理できるようにすることです。これにより、コードの重複を減らし、保守性や可読性を向上させます。

11.1 構成要素

Spring AOP では、以下の 5 つの要素が基本的な構成要素です：

11.1.1 アスペクト (Aspect)

横断的な機能そのものです。たとえば、ログ出力や例外処理をひとつのアスペクトとして実装します。

11.1.2 アドバイス (Advice)

アスペクトがいつ実行されるかを定義します。メソッドの実行前、実行後、例外発生時など、どのタイミングで横断的な処理を行うかを指定します。

11.1.3 ジョインポイント (Join Point)

アスペクトが適用される具体的なポイントです。Spring では、主にメソッド呼び出しがジョインポイントとなります。

11.1.4 ポイントカット (Pointcut)

ジョインポイントをフィルタリングし、どのメソッドにアドバイスを適用するかを定義します。正規表現やアノテーションを使って、対象となるメソッドを柔軟に指定できます。

11.1.5 ターゲット (Target)

アスペクトを適用されるオブジェクト、つまりビジネスロジックを持つクラスです。

11.2 簡単な例

たとえば、特定のメソッドが実行される前にログを出力するアスペクトを作成する場合、以下の流れとなります：

- (1) @Before アノテーションを使って、メソッドの実行前にログ出力を行うアドバイスを定義。
- (2) ポイントカットで、どのメソッドに対してこのアドバイスを適用するかを指定。

@Aspect

@Component

```
public class LoggingAspect {  
    @Before("execution(* com.example.service.*(..))")  
    public void logBefore(JoinPoint joinPoint) {
```

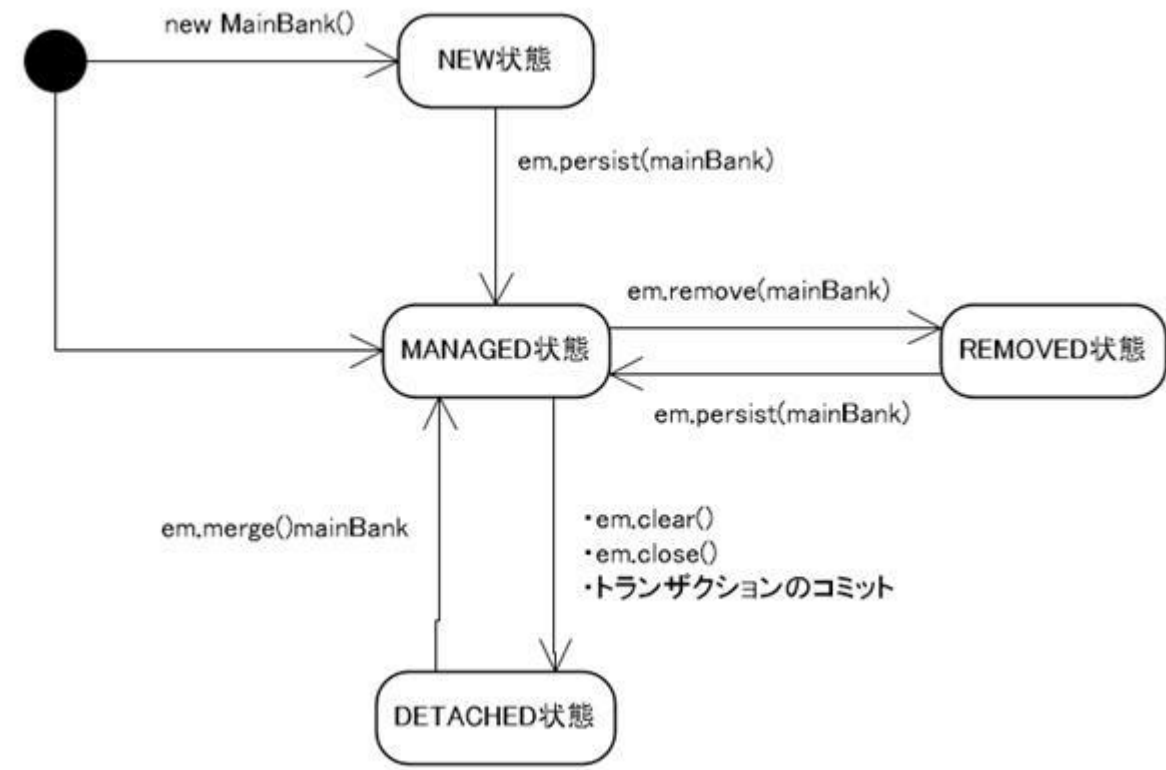
```
        System.out.println("Executing method: " + joinPoint.getSignature());  
    }  
}
```

このコードでは、com.example.service パッケージ内のすべてのメソッドが実行される前に、メソッド名をログに出力します。

Spring AOP を使用すると、ビジネスロジックに直接関与しないコードを分離し、関心ごとに応じた処理を柔軟に追加できるようになります。

12. Spring JPA 永続化

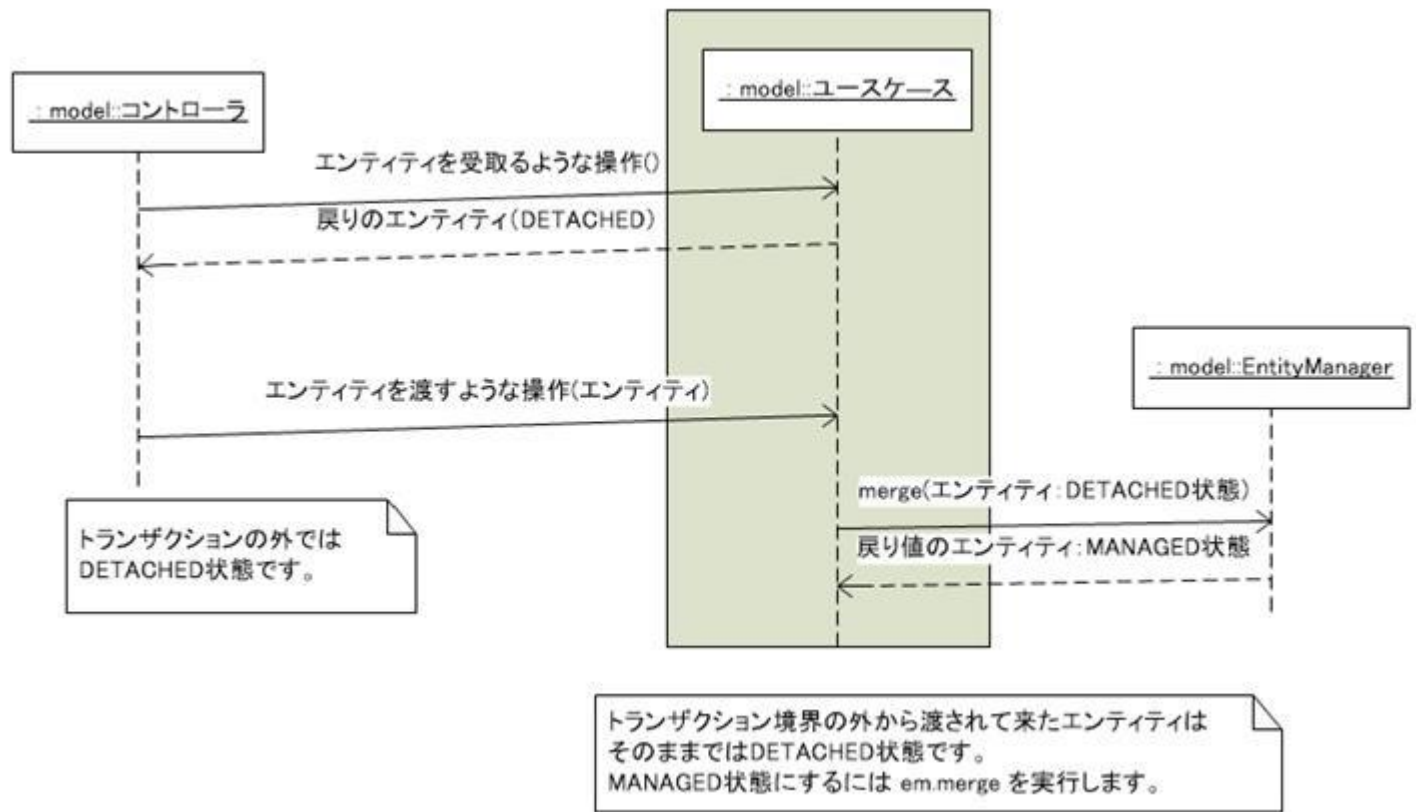
12.1 エンティティオブジェクトの状態遷移



12.2 エンティティオブジェクトの状態

| 状態名 | 概要 |
|-------------|---|
| NEW 状態 | エンティティオブジェクトを生成（new）しただけでの状態です。永続化コンテキストとは関連付けられていません。 |
| MANAGED 状態 | エンティティオブジェクトは永続化コンテキストと関連付けられた状態です。EntityManager#find で検索可能です。 |
| DETACHED 状態 | エンティティオブジェクトは永続化コンテキストと関連付けられていません。 MANAGED 状態から DETACHED 状態への遷移は EntityManager#clear または EntityManager#close で操作できます。 もう一つ、トランザクションに関係した挙動でも遷移します（後述：merge 参照） |
| REMOVED 状態 | 削除された状態です。EntityManager#persist で再び MANAGED 状態に遷移できます。 |

12.3 トランザクション境界での状態遷移の挙動



- merge メソッドの API doc

<T> T merge(T entity)

Merge the state of the given entity into the current persistence context.

Parameters:

entity - entity instance

Returns:

the managed instance that the state was merged to

Throws:

IllegalArgumentException - if instance is not an entity or is a removed entity

TransactionRequiredException - if there is no transaction when invoked on a container-managed entity manager of that is of type PersistenceContextType.TRANSACTION

13. MySQL

MySQL80 と MySQLWorkbench8.0 を前提に説明します。

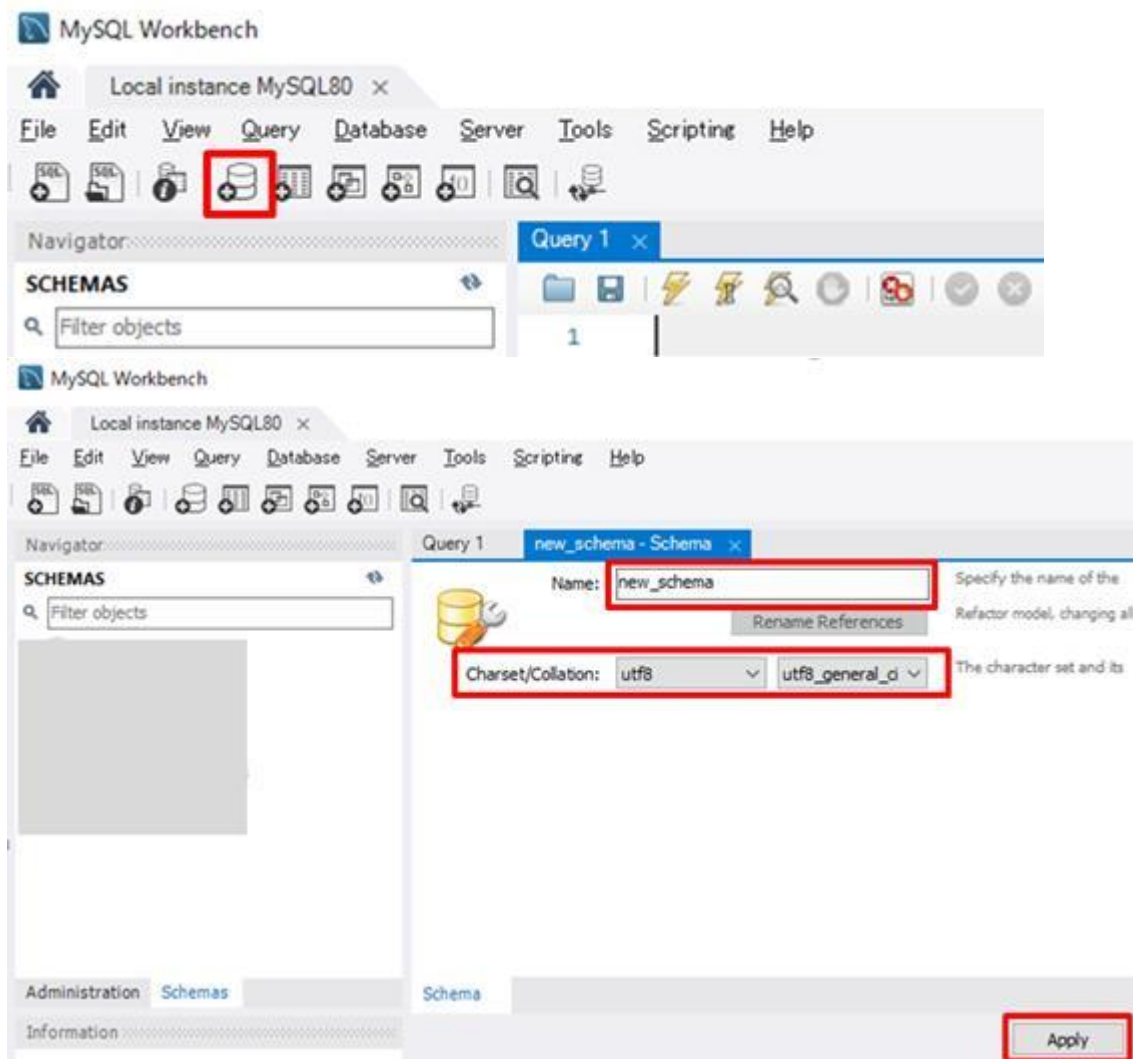
MySQL では「データベース」と「スキーマ」を区別しません。本書では同意語として使います。

13.1 データベースの作成

(作成例)



- (1) [create a new schema]というボタンをクリック (赤枠内をクリック)
- (2) Name には、登録したいスキーマの名前を入力
- (3) charset (文字セット) /collation (デフォルトの照合順序) には、utf8/utf8_general_ci を選択
- (4) Apply で確定



13.2 ユーザとパスワードの作成

(作成例)



13.2.1 username と password の確認方法と設定方法

(1) MySQL Workbench でユーザー情報を確認する方法:

MySQL Workbench を開き、MySQL インスタンスに接続します。

メインメニューから「Server (サーバー)」 -> 「User and Privileges (ユーザーと権限)」をクリックします。

左側のユーザーリストから確認したいユーザー名 (例: mainbank) を選択します。

右側に選択したユーザーの詳細 (認証方式、パスワード、権限など) が表示されます。

(2) ユーザー名とパスワードの設定方法:

上記の「User and Privileges」画面で、パスワードを変更したい場合は、「Login (ログイン)」タブで password フィールドを変更し、適用します。

新しいユーザーを追加したい場合は、同じ画面の「Add Account (アカウントを追加)」ボタンを押し、新しいユーザー名とパスワードを設定し、適切な権限を割り当てます。

13.3 application.properties の設定

(設定例)

spring.application.name=mainbank

spring.thymeleaf.mode=HTML

spring.datasource.url=jdbc:mysql://localhost:3306/db_mainbank?serverTimezone=UTC

```
spring.datasource.username=mainbank
spring.datasource.password=mainbank
# spring.datasource.driver-class-name =com.mysql.jdbc.Driver => 'com.mysql.jdbc.Driver'. This is
deprecated.
spring.datasource.driver-class-name =com.mysql.cj.jdbc.Driver
spring.jpa.database=MYSQL
# create-drop cause Error executing DDL "alter table
#none / validate / update / create / create-drop / drop
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
spring.jpa.show-sql: true
spring.security.user.name=test
spring.security.user.password=test
#platform line seems unnecessary
#spring.datasource.platform=mysql
```

13.4 データベースとスキーマ

MySQL の場合、データベースとスキーマはほぼ同じ意味で使われます。

- データベース: MySQL では「データベース」は、テーブルやビューなどのデータオブジェクトが保存される領域のことを指します。
- スキーマ: 一部のデータベースシステムでは、スキーマはデータベース内でテーブルや他のオブジェクトの構造を定義する「論理的なコレクション」です。

13.5 spring.jpa.hibernate.ddl-auto の設定

Spring Boot アプリケーションにおける Hibernate のデータベーススキーマの自動生成や変更に関する設定です。これには複数のオプションがあり、それぞれ異なる挙動を持っています。

13.5.1 代表的なオプション

(1) none

挙動: Hibernate がスキーマ管理を一切行わず、データベースの構造はそのままです。

用途: スキーマの自動生成や変更を避けたい場合。既存のスキーマがあり、手動で管理したいときに使用します。

(2) validate

挙動: エンティティの定義とデータベースのスキーマが一致しているかを検証しますが、データベースに対して何も変更を加えません。

用途: スキーマが正しく設定されているか確認したい場合や、変更を加えずにスキーマの整合性をチェックしたい場合に適しています。

(3) update

挙動: エンティティの定義に基づいて、必要なテーブルやカラムを作成・更新します。ただし、既存のデータや構造を削除することはありません。

用途: スキーマを自動的に変更したい場合や、既存のデータを維持したまま、変更を反映したい場合に適しています（現状これを使用中です）。

(4) create

挙動: アプリケーション起動時に既存のスキーマをすべて削除し、新しくスキーマを作成します。アプリケーション終了時にはスキーマは削除されません。

用途: 開発環境で新しいスキーマをすぐに作りたい場合に使用されますが、既存データは消えるため注意が必要です。

(5) create-drop

挙動: アプリケーションの起動時にスキーマを作成し、終了時にスキーマを削除します。

用途: テストや一時的な環境で、スキーマを簡単に作成し、使用後にクリーンアップしたい場合に使用します。

(6) drop

挙動: アプリケーション起動時にスキーマを削除し、その後何もしません。

用途: 一般的にはあまり使用されませんが、起動時に明示的にスキーマを削除したい場合に使用します。

13.5.2 オプションのまとめ

none: スキーマ操作をしない（手動管理）。

validate: スキーマの検証のみ（変更なし）。

update: スキーマの更新（データは保持）。

create: スキーマを再作成（データは削除）。

create-drop: 起動時にスキーマを作成し、終了時に削除。

drop: スキーマを削除。

update は開発時に便利ですが、本番環境では validate や none を使って、スキーマを手動で管理するのが一般的です。

14. Tips レンダリング

レンダリングとは、Web アプリ開発の文脈では、クライアントサイド・レンダリングやサーバサイド・レンダリングのように HTML データを生成する処理を指します。

また、最終的に、ブラウザがデータ（HTML など）を視覚化する処理を指す場合もあります。

14.1 サーバサイドレンダリング（Server-Side Rendering, SSR）

JSP（JavaServer Pages）はサーバサイドレンダリングの一種です。

サーバサイドレンダリングとは、サーバー側で HTML を生成してクライアント（ブラウザ）に送信する方法です。

JSP は Java でサーバーサイドのコードを実行し、その結果を HTML として生成します。具体的には、JSP ページは Java コードや JSP タグを含んでおり、サーバー上で実行されると、その出力が HTML として生成され、クライアントに送信されます。

JSP のレンダリングプロセスは次のようになります：

- (1) クライアントが JSP ページにリクエストを送信する。
- (2) サーバーが JSP ファイルを受け取り、Java Servlet としてコンパイルする。
- (3) コンパイルされた Servlet が実行され、HTML 出力を生成する。
- (4) 生成された HTML がクライアントに送信され、ブラウザで表示される。

このように、JSP はサーバーサイドで実行されるため、サーバーサイドレンダリングの一部とされています。

14.2 クライアントサイドレンダリング (Client-Side Rendering, CSR)

(1) JavaScript フレームワーク

React, Angular, Vue.js などの JavaScript フレームワークはクライアントサイドレンダリングを前提として設計されています。これらのフレームワークは、コンポーネントベースのアーキテクチャを使用して、クライアント側で効率的に UI を構築・更新します。

(2) SPA (Single Page Application)

クライアントサイドレンダリングは、一般的に SPA アプリケーションで使用されます。SPA では、最初に必要な HTML、CSS、JavaScript をロードし、その後のページ遷移は JavaScript によって行われ、サーバーから新しい HTML を取得するのではなく、必要なデータだけをフェッチして、ページを動的に更新します。

14.3 静的サイト生成 (Static Site Generation, SSG)

ウェブサイトのページを事前に生成し、それらを静的ファイル（通常は HTML、CSS、JavaScript など）としてサーバーに配置する手法です。SSG は、あらかじめ定義されたテンプレートとデータを基に、ビルド時に全てのページを生成するので、リクエスト時にサーバー側で動的に生成する必要がなく、非常に高速で効率的です。

15. Tips : セレクトボックスの書き方

- HTML 素のコード

```
<select name="example">
<option value="サンプル 1">サンプル 1</option>
<option value="サンプル 2">サンプル 2</option>
<option value="サンプル 3">サンプル 3</option>
</select>
```

- JSP JSTL を使うとき

```
<c:forEach items="${itemArray}" var="item" varStatus="status">
```

```
<option value="${status.count}" <c:if test="${status.count==itemIndexStr}">selected</c:if>>
${item}</option>
</c:forEach>
```

- Spring (Spring MVC) を利用したとき

```
<!--職種 -->
```

```
<div class="mr-2">
```

```
<form:select path="jobTypeId" onchange="showHome()">
```

```
<form:options items="${jobTypeList}" itemLabel="label" itemValue="id" /
```

```
</form:select>
```

```
</div>
```

- Thymeleaf を利用したとき

```
<select class="dropdown form-control">
```

```
<option th:value="0"></option>
```

```
<option th:each="item : ${mDepartmentMap}" th:value="${item.value}"
th:text="${item.key}"></option>
```

```
</select>
```