Section

1-1 Spring Frameworkの全体像を知ろう

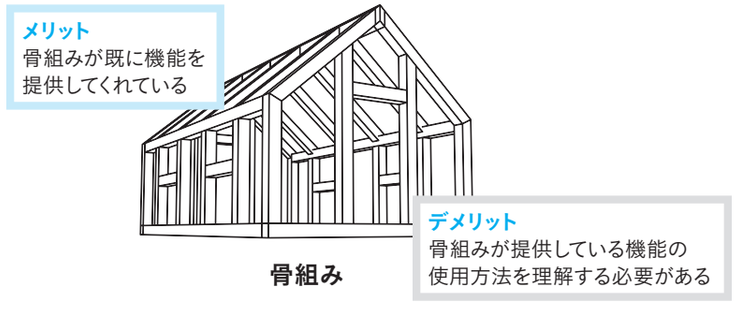
この章では、現時点でJavaの主要フレームワークと言われている「Spring Framework」について、「Java基礎文法を理解している方」を対象に、なるべく簡単に説明します。その後、本書で実施するハンズオンの開発環境構築について説明します。この章を読み終わった時に「Spring Frameworkのイメージ」を何となく掴んで頂けたら幸いです。

1-1-1 フレームワークとは？

まず、フレームワークとは何でしょうか？フレームワークとは、簡単に言うとソフトウェアやアプリケーション開発を行う事を簡単にする「骨組み」です（**図1.1**）。フレームワークのメリットとして、フレームワークが必要最低限の機能を提供してくれるため、自分ですべての機能を作成する必要がなく、アプリケーションの開発にかかる時間とコストを削減できます。

デメリットとして、フレームワークを利用する開発では、フレームワーク特有の使用方法を理解する必要があります。

図1.1 「フレームワーク」のイメージ



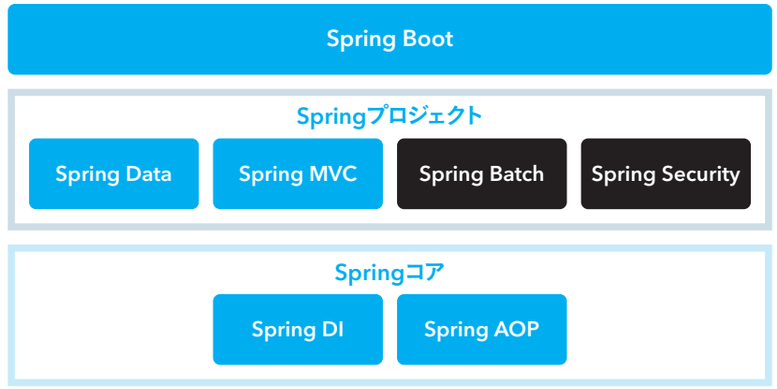
1-1-2 Spring Frameworkとは？

「Spring Framework」は、Java開発におけるフレームワークです。単に「Spring」とも呼ばれます（**図1.2**）。

開発が楽になる様々な機能を提供してくれていて、機能毎にプロジェクトが存在しています。

以下にSpringが提供してくれている機能の一部について紹介します。

図1.2 「Spring Framework」の全体像



Spring Boot

Springアプリケーションを煩雑な設定をせず迅速に作成する機能を提供します。

Springプロジェクト

- Spring MVC
  Webアプリケーションを簡単に作成する機能を提供します。
- Spring Data
  データアクセスに対する機能を提供します。
- Spring Batch
  バッチ処理機能を提供します。
- Spring Security
  認証/認可の機能を提供します。

Springコア

- Spring DI
  依存性注入の機能を提供します。
- Spring AOP
  アスペクト指向プログラミングの機能を提供します。

本書で扱う機能は、「Spring Boot」、「Spring MVC」、「Spring Data」、「Spring DI」、「Spring AOP」になります。詳しい説明は対応する各章にて説明しますので、現時点では「Spring Framework」とは様々な機能を提供してくれるフレームワークだとイメージしてください。

Section

3-2 DIコンテナについて知ろう

DIコンテナとは「Dependency Injection：依存性の注入」略して「DI」の実現をお手伝いするためのフレームワークです。DIコンテナを説明する前に、Javaにおける依存性について知る必要があります。

3-2-1 依存性

まずプログラムには「使う側」と「使われる側」という関係があります（**図3.3**）。

図3.3 「使う側」と「使われる側」のイメージ

使いたい機能を呼び出すには「使う側」クラスで「使われる側」クラスに対し「new」キーワードを使用してインスタンスを生成し参照を取得後、目的の機能を利用します。もし「使われる側」クラスが不要になって別の「使われる側」クラスを利用する場合、「使う側」クラスで「新たな使われる側」のクラス名及びメソッド名に書き換える必要があります。書き換える箇所を「依存性がある」といいます。

依存には以下2種類があります。

- クラス依存（実装依存）
- インターフェース依存

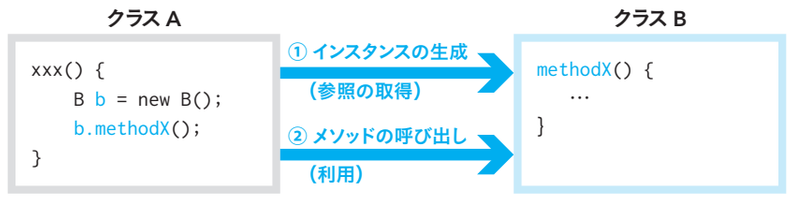
3-2-2 クラス依存

まずはクラス依存から説明します。

「使う側」クラスAで「使われる側」クラスBのメソッド「methodX」を呼んでみましょう（**図3.4**）。

- クラスAでnewキーワードを使用してクラスBインスタンスを生成します
- インスタンスからメソッド「methodX」を呼び出します

図3.4 クラス依存「インスタンスの生成」と「メソッドの呼び出し」

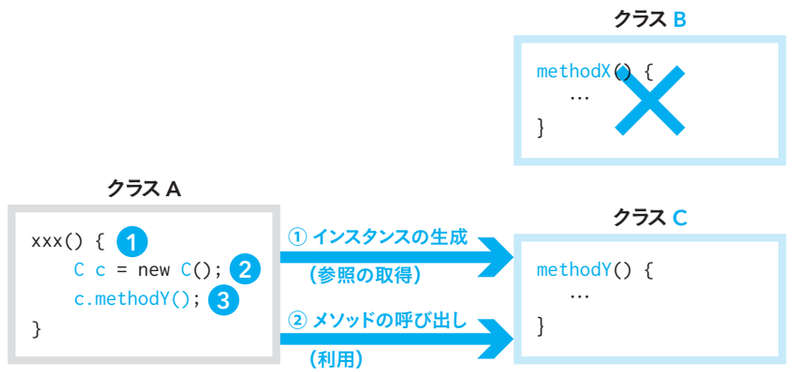


仕様変更があり、「使われる側」クラスを変更することになりました（**図3.5**）。新たに作成された「使われる側」クラスCを呼び出し、メソッド「methodY」を呼び出すように変更してください。

図3.5 クラス依存「使われる側」クラスの変更

みなさんなら、どの様にクラスAのメソッド「xxx」を修正しますか？筆者なら以下の様に修正します（**図3.6**）。

図3.6 クラス依存「使う側」クラスの3箇所の修正



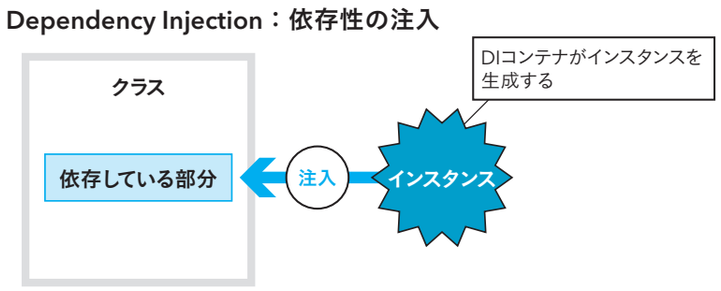
3-2-5 DIコンテナ

依存性の注入とは「依存している部分を、外から注入すること」でした。
言葉を分解すると以下になります。

- 「依存している部分」とは「使う側」クラスに「使われる側」クラスが記述されている状態
- 「外から注入」とは、「使う側」クラスの外から「使われる側」インスタンスを注入すること

今まではインスタンス生成には「new」キーワードを使用していましたが、インスタンス生成は面倒なのですべてフレームワークに任せたいです。その責務を引き受けてくれるのが「DIコンテナ」になります（**図3.22**）。Spring Frameworkは任意に実装したクラスをインスタンス化する機能を提供しています。つまりDIコンテナの機能を持っています。

図3.22 「Dependency Injection」イメージ



3-2-6 5つのルール

DIコンテナにインスタンス生成を任し、以下のルールを守ることで「使う側」クラスの修正を「ゼロ」にすることができます。

① インターフェースを利用して依存関係を作る
② インスタンスを明示的に生成しない
③ アノテーションをクラスに付与する
④ Spring Frameworkにインスタンス生成させる
⑤ インスタンスを利用したい箇所でアノテーションを付与する

ではルールについて順番に説明します。

ルール①

「インターフェースを利用して依存関係を作る」とは、「依存している部分」には「インターフェース」を利用することです。

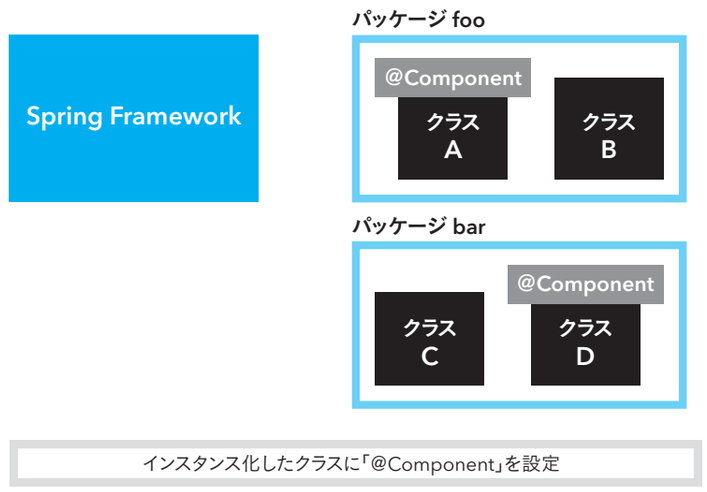
ルール②

「インスタンスを明示的に生成しない」とは、インスタンス生成に「new」キーワードを利用しないということです。

ルール③とルール④

「アノテーションをクラスに付与する」と「Spring Frameworkにインスタンス生成させる」をまとめて説明します。インスタンス化したいクラスにインスタンス生成アノテーションを付与します。アノテーションについての詳細は「**3-3 アノテーションの役割を知ろう**」で説明しますが、ここでは「目印」の様なものだとイメージしてください。**図3.23**の「@Component」をアノテーションといいます。

図3.23 「@Component」を付与する



Spring Frameworkは「起動時」に対象プロジェクトのパッケージをすべてスキャンします。この機能を「コンポーネントスキャン」といいます（**図3.24**）。

同様に、「Greet」インターフェースの実装クラス「EveningGreet」の内容は**リスト3.8**のようになります。10～12行目「greeting」メソッドの処理内容（夕方の挨拶）を記述します。

リスト3.8 実装クラス「EveningGreet」

```
001: package com.example.demo.chapter03.used;
002:
003: /**
004:  * Greet実装クラス<br>
005:  * 夕方の挨拶を行う
006: */
007: public class EveningGreet implements Greet {
008:     @Override
009:     public void greeting() {
010:         System.out.println("------------------");
011:         System.out.println("こんばんは。");
012:         System.out.println("------------------");
013:     }
014: }
```

04 ルール③④の実施

ルール③で示した「アノテーションをクラスに付与する」とルール④「Spring Frameworkにインスタンス生成させる」を実施します。具体的には「Greet」インターフェースを実装したクラス「MorningGreet」に「@Component」を付与します。インポート宣言に「import org.springframework.stereotype.Component;」が追加されます。

インポートがされない場合は、eclipseのショートカットキー「Ctrl + Shift + o（オー）」を使用して「import文の編成」を実施してください。

```
@Component
public class MorningGreet implements Greet {
```

05 「使う側」クラスの作成

「Springスターター・プロジェクト」でプロジェクトを作成すると、デフォルトで自分が作成した「プロジェクト名 + Application」クラスが作成されます（**リスト3.9**）。このクラスには「@SpringBootApplication」アノテーションが付与されています。メインメソッドを含んだクラスに「@SpringBootApplication」アノテーションを付与すると「SpringBootアプリケーション」と認識されます。

ここではクラス「DependencyInjectionSampleApplication」はSpringBootアプリケーションの「起動クラス」とだけイメージしてください。

リスト3.9 使う側クラス「DependencyInjectionSampleApplication」

```
001: package com.example.demo;
002:
003: import org.springframework.boot.SpringApplication;
004: import org.springframework.boot.autoconfigure.SpringBootApplication;
005:
006: /**
007:  * SpringBoot起動クラス
008:  */
009: @SpringBootApplication
010: public class DependencyInjectionSampleApplication {
011:     public static void main(String[] args) {
012:         SpringApplication.run(DependencyInjectionSampleApplication.class, args);
013:     }
014: }
```

06 ルール①②⑤の実施

ルール①、②、⑤で示した「インターフェースを利用して依存関係を作る」「インスタンスを明示的に生成しない」「インスタンスを利用したい箇所でアノテーションを付与する」の3つを実施します。

具体的には「Spring Framework」によって生成されたインスタンスを利用したい箇所に、参照を受け取る「フィールド」を宣言し、「フィールド」に「@Autowired」アノテーションを付与します。ここでは「使う側」クラス「DependencyInjectionSampleApplication」に「使われる側」インターフェース「Greet」をフィールドに宣言し、変数に「@Autowired」アノテーションを付与します。

```
/**
 * 注入される箇所(インターフェース)
 */
@Autowired
Greet greet;
```

「使われる側」インターフェース「Greet」のメソッド「greeting」を実行するメソッドを作成します。メソッド名は「execute」とします（**リスト3.10**）。

リスト3.10 「Greet」インターフェースのメソッド「greeting」を実行するメソッド

```
001: /**
002:  * 実行メソッド
003:  */
004: private void execute() {
005:     greet.greeting();
006: }
```

Section

4-4 「エンティティ」と「リポジトリ」を知ろう

SQLを使用してテーブルへの「CRUD」操作を説明しました。ここでは「プログラムからのデータベース操作」に関連する用語「Entity（エンティティ）」と「Repository（リポジトリ）」について説明します。

4-4-1 Entity（エンティティ）とは？

エンティティを一言で表現するなら「データの入れ物となるクラス」です。もう少し詳細に説明するとエンティティは、データベースの「テーブルの１行」に対応するクラスです（**図4.16**）。「エンティティ」が保持するフィールドは、テーブルのカラム値に対応します（**リスト4.1**）。

図4.16 「エンティティ」と「テーブル」の対応関係

リスト4.1 「Member」エンティティの例

```
001: /**
002:  * Memberテーブル：エンティティ
003:  */
004: public class Member {
005:     /** idカラムに対応 */
006:     private Integer id;
007:     /** nameカラムに対応 */
008:     private String name;
009:
010:     public Integer getId() {
011:         return id;
012:     }
013:     public void setId(Integer id) {
014:         this.id = id;
015:     }
016:     public String getName() {
017:         return name;
018:     }
019:     public void setName(String name) {
020:         this.name = name;
021:     }
022: }
```

「エンティティ」はただの「データの入れ物」ですが、使用方法の特徴が３つあります。

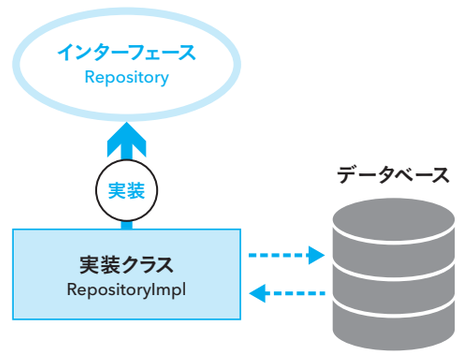
- クラス名
  エンティティのクラス名は、対応するデータベースのテーブル名と同じにすることが多いです
- データベースへ値の引き渡し
  データベースへ値を登録・更新する場合に、エンティティに値を入れて引き渡します
- データベースから値の所得
  データベースから値を取得した場合に、値をエンティティに入れて保持しておきます

4-4-2 Repository（リポジトリ）とは？

リポジトリとは簡単に言うと「データベースへのデータ操作」を行うクラスです。リポジトリを作成する場合には、必ずインターフェースを定義したうえで実装します。その理由は「使用する側」の「リポジトリ」インターフェースの「フィールド」に「リポジトリ」実装クラスを「DI」することで、特定の実装への依存を回避することができるからです（**図4.17**）。

依存性については「**3-2 DI コンテナについて知ろう**」で説明しています。 なお、Javaではインターフェースを実装したクラス名の接尾辞に「Impl」とつけることが多いです（「Impl」は「implements」の略になります）。

図4.17 Repositoryのイメージ



Section 6-3

Thymeleafの使い方

ここでは、「Thymeleaf」とは何かを再度確認した後、具体的な使用方法を説明します。その後、説明した「Thymeleaf」の使用方法を使って、「コントローラ」から「ビュー」を表示させるプログラムを作成しましょう。

6-3-1 Thymeleafの復習

「Thymeleaf」の特徴を簡単に説明すると以下の3点になります。

- 「Spring Boot」で使用が推奨されている「テンプレートエンジン」です
- 「HTML」としてテンプレートを記述することができるため、Webブラウザでファイル内容を表示させ、確認しながら「ビュー」を作成することができます
- 「Spring Boot」のデフォルト設定では、「src/main/resources/templates」フォルダ配下の「リクエストハンドラメソッドの戻り値＋.html」ファイルが参照されます

6-3-2 Thymeleafの使用方法

直接文字を埋め込む

直接文字を埋め込む例を**リスト6.3**に示します。

リスト6.3 直接文字を埋め込む

```
001:  <!-- 01：直接文字を埋め込む -->
002:  <h1 th:text="hello world">表示する部分</h1>
```

「th:text="【出力文字】"」とすることで、設定した文字を出力できます。また【出力文字】の部分は「Thymeleaf」独自式「${}」を使用できます。属性値の値設定で「"（ダブルクォーテーション）」を使用しているので、文字を設定する時は、「'（シングルクォーテーション）」で囲みます。

インライン処理

インライン処理の例を**リスト6.4**に示します。

リスト6.4 インライン処理

```
001:  <!-- 02：インライン処理 -->
002:  <h1>こんにちは！[[${name}]]さん</h1>
```

[[${}]]を使用すると、タグの属性への追加ではなく本体へ変数を埋め込めます。「固定値」と「変数」を組み合わせたい場合には、こちらの方法が便利です。

値結合

値結合の例を**リスト6.5**に示します。

リスト6.5 値結合

```
001:  <!-- 03：値結合 -->
002:  <h1 th:text="'明日は、' + '晴れ' + 'です。'">表示する部分</h1>
```

「＋」を利用して「値」の連結ができます。

値結合（リテラル置換）

値結合（リテラル置換）の例を**リスト6.6**に示します。

リスト6.6 値結合（リテラル置換）

```
001:  <!-- 04：値結合（リテラル置換） -->
002:  <h1 th:text="|こんにちは！${name}さん|">表示する部分</h1>
```

値結合はリテラル置換を使用することで、"| 文字 |"で記述ができます。文字の中で「${}」式も合わせて使用できます。

ローカル変数

ローカル変数の例を**リスト6.7**に示します。

リスト6.7 ローカル変数

```
001:  <!-- 05：ローカル変数 -->
002:  <div th:with="a=1,b=2">
003:      <span th:text="|${a} + ${b} = ${a+b}|"></span>
004:  </div>
```

Section

7-3 入力値を受け取るプログラムを作成しよう（Formクラス）

「@RequestParam」アノテーションは便利ですが、リクエストパラメータを個々に引数で受け取るため、入力項目が増えれば増えるほど引数が増えてしまい冗長になります。「Spring MVC」では、「入力値を格納するためのクラス」を用意することで、リクエストパラメータを「まとめて」引き渡すことができます。先ほど作成したプログラムに「入力値を格納するためのクラス」を用意してみましょう。

7-3-1 Formクラスの作成

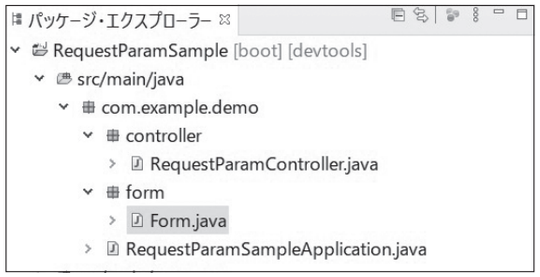
「入力値を格納するためのクラス」→「入力項目はビュー側でformタグ内に囲まれている」→「Formクラス」という「ビュー」のフォームを表現するクラスを作成します。

「src/main/java」→「com.example.demo」フォルダを選択し、マウスを右クリックし、「新規」→「クラス」を選択します。

パッケージを「com.example.demo.form」名前を「Form」としてクラスを作成します（**図7.6**）。

Formクラスの内容は**リスト7.5**になります。

図7.6 Formクラス



リスト7.5 Formクラス

```
001: package com.example.demo.form;
002:
003: import java.time.LocalDate;
004:
005: import org.springframework.format.annotation.DateTimeFormat;
006:
007: import lombok.Data;
008:
009: @Data
010: public class Form {
011:     private String name;
012:     private Integer age;
013:     @DateTimeFormat(iso = DateTimeFormat.ISO.DATE)
014:     private LocalDate birth;
015: }
```

9行目「Lombok」の機能を使用して「getter/setter」を「@Data」で生成します。13行目「@DateTimeFormat」で「iso = DateTimeFormat.ISO.DATE」と指定し「日付形式 yyyy-MM-dd」で受け取る様に指定します。

01 「コントローラ」への修正・追記

「コントローラ：RequestParamController」にリクエストハンドラメソッドを修正・追記します（**リスト7.6**）。

リスト7.6 RequestParamControllerの追記

```
001:     /** 確認画面を表示する */
002: //  @PostMapping("confirm")
003: //  public String confirmView(
004: //      Model model, @RequestParam String name, @RequestParam Integer age,
005: //      @DateTimeFormat(iso = DateTimeFormat.ISO.DATE) @RequestParam LocalDate
     birth
006: //   ) {
007: //      // Modelに格納する
008: //      model.addAttribute("name",name);
009: //      model.addAttribute("age",age);
010: //      model.addAttribute("birth",birth);
011: //      // 戻り値は「ビュー名」を返す
012: //      return "confirm";
013: //     }
014:     /** 確認画面を表示する：Formクラス使用 */
015:     @PostMapping("confirm")
016:     public String confirmView(Form f) {
017:         // 戻り値は「ビュー名」を返す
018:         return "confirm2";
019:     }
```

2～13行目既に作成しているリクエストハンドラメソッド「confirmView」をコメントアウトします。

15～19行目「@PostMapping("confirm")」で「POSTかつURL：/confirm」に対応するリクエス

Section

8-1 バリデーションの種類を知ろう

7章までで「リクエストパラメータの取得について」イメージできましたでしょうか。「ビュー」で入力を行う時に、疑問に思う内容として「数値を入力してもらいたいのに、文字列を入力された時などはどうなるのか?」という疑問です。この章では「ビュー」で入力した値に対して「入力チェック」を行う「バリデーション」機能について説明します。

8-1-1 バリデーションとは?

「バリデーション」とは、入力内容が要件を満たしているか、妥当性を確認する「入力チェック」のことです。バリデーションを大きく分けると以下の2つに分かれます。

- 単項目チェック
- 相関項目チェック(相関チェック)

8-1-2 単項目チェックとは?

「単項目チェック」とは、入力項目1つ1つに対して、設定する「入力チェック」機能です。使用方法はFormクラスなどのフィールドに「アノテーション」を付与します。

「入力チェック」の「アノテーション」は、Java EEが提供するアノテーション「Bean Validation」やHibernateフレームワークが提供するアノテーション「hibernate Validator」があります。また数値入力項目にアルファベットなどの文字列を入力した場合などの「型変換チェック」は「入力チェック」機能を有効にするだけで利用できるためアノテーションの記述はいりません。

表8.1に単項目チェックで利用される主な「アノテーション」を記述します。

表8.1 単項目チェックで利用される主なアノテーション

アノテーション	機能概要	使用例
@NotNull	nullでないことを検証します	`@NotNull` `Integer no;`
@NotEmpty	文字列がnull又は空文字("")でないか検証します	`@NotEmpty` `String name;`
@NotBlank	文字列がnull又は空白スペース(半角スペースやタブなど)でないか検証します	`@NotBlank` `String name;`

@Max	指定した数値以下であることを検証します	`#100以下であることを検証`
		`@Max(100)`
		`Integer price;`

@Min	指定した数値以上であることを検証します	`#10以上であることを検証`
		`@Min(10)`
		`Integer age;`

@Size	文字列やCollectionが指定した範囲の大きさであることを検証します	`#要素数が0から10の範囲か検証(文字列は文字列長、Collectionはサイズ)`
		`@Size(min=0,max=10)`
		`List<Integer> selected;`

@AssertTrue	trueであることを検証します	`@AssertTrue` `Boolean empty;`
@AssertFalse	falseであることを検証します	`@AssertFalse` `Boolean empty;`

@Pattern	指定された正規表現に一致するか検証します	`#半角英数字か検証`
		`@Pattern(regexp = "[a-zA-Z0-9]*")`
		`String alphaNum;`

@Range	指定された数値の範囲内であるか検証します	`#1以上10以下であるか検証`
		`@Range(min=1,max=10)`
		`Integer point;`

@DecimalMax	指定された数値以下であるか検証します	`#100.0以下であることを検証(小数部を含む場合はMaxでなくDecimalMaxを使用)`
		`@DecimalMax("100.0")`
		`BigDecimal val;`

@DecimalMin	指定された数値以上か検証します	`#10.0以上であることを検証(小数部を含む場合はMinではDecimalMinを使用)`
		`@DecimalMin("10.0")`
		`BigDecimal val;`

@Digits	整数部と小数部の桁数を検証します	`#整数部が3、小数部が1であるか検証`
		`@Digits(integer = 3, fraction = 1)`
		`BigDecimal val;`

@Future	未来の日付か検証します	`@Future` `Date date;`

Section

9-1 作成するアプリケーションの説明

ここから1章～8章までに学習した内容を使用し、Webアプリケーションを作成します。作成する内容は「○×クイズ」アプリです。「○×クイズ」とは「出題されたクイズに対して、○か×で解答する形式の設問」です。ここでは「○×クイズ」アプリを作成する準備として機能の説明やデータベース、プロジェクトの作成を行います。

9-1-1 機能一覧

「○×クイズ」アプリの機能は、**表9.1**にまとめた5つになります。「No.1～4」は「CRUD」処理です。「No.5」は登録したクイズをランダムで表示し、解答する機能です。

表9.1 作成機能一覧

No	機能	説明
1	登録機能	クイズを登録します
2	更新機能	登録したクイズを変更します
3	削除機能	登録したクイズを削除します
4	一覧表示	登録したクイズを一覧表示します
5	ゲーム機能	クイズを解きます

「○×クイズ」アプリのURLに対する役割を**表9.2**に記述します。

表9.2 URL一覧

No	役割	HTTPメソッド	URL
1	一覧画面を表示する	GET	/quiz
2	登録処理を実行する	POST	/quiz/insert
3	更新画面を表示する	GET	/quiz/{id}
4	更新処理を実行する	POST	/quiz/update
5	削除処理を実行する	POST	/quiz/delete
6	クイズ画面を表示する	GET	/quiz/play
7	クイズを解く	POST	/quiz/check

9-1-2 作成するアプリケーションのレイヤ化

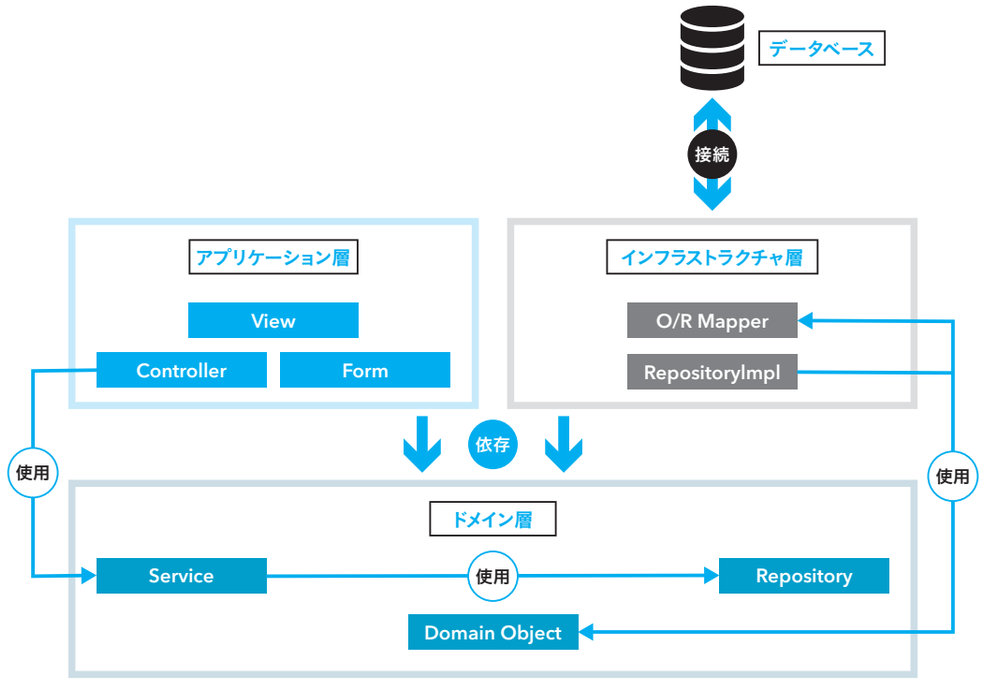
「**3-3-2 レイヤ別で使用するインスタンス生成アノテーション**」でアプリケーションを作成する時、レイヤで分ける事が推奨されていることを説明しました。今回作成する「○×クイズ」アプリもレイヤに分けて開発します。 「○×クイズ」アプリは、次の3レイヤで分割します。

- アプリケーション層
- ドメイン層
- インフラストラクチャ層

アプリケーション層、ドメイン層、インフラストラクチャ層はEric Evansの「Domain Driven Design：ドメイン駆動設計」略して「DDD」で説明されている用語です。本書では用語は使用していますが、「DDD」の考えにのっとっているわけではありません。

「**5-1 MVCモデルについて知ろう**」で説明した「MVCモデル」ですが、業務機能や扱うデータ要件が複雑になるほど「サービス処理内容を記述する」Model（モデル：M）の担当する部分が多くなってしまい「Modelの肥大化」という問題が発生します。「MVCモデル」の設計上、「Model」が担う役割自体を減らすことはできないので「Model」の中の「役割分担」をより明確に、アプリケーションのレイヤ構成を当てはめて、肥大化するModelを分割しようというのがレイヤ化の考え方になります。 各レイヤには、**図9.1**に示すコンポーネント（部品）が含まれます。

図9.1 レイヤ別コンポーネント



Section

12-1 アプリケーション層を確認しよう

ここでは「アプリケーション層」の作成を行います。「画面」や「リクエストハンドラメソッド」などを作成し、「〇×クイズ」アプリを完成させましょう。

12-1-1 作成部分の確認

この章で作成する箇所は、**図12.1**の点線枠になります。

図12.1 レイヤ別コンポーネント

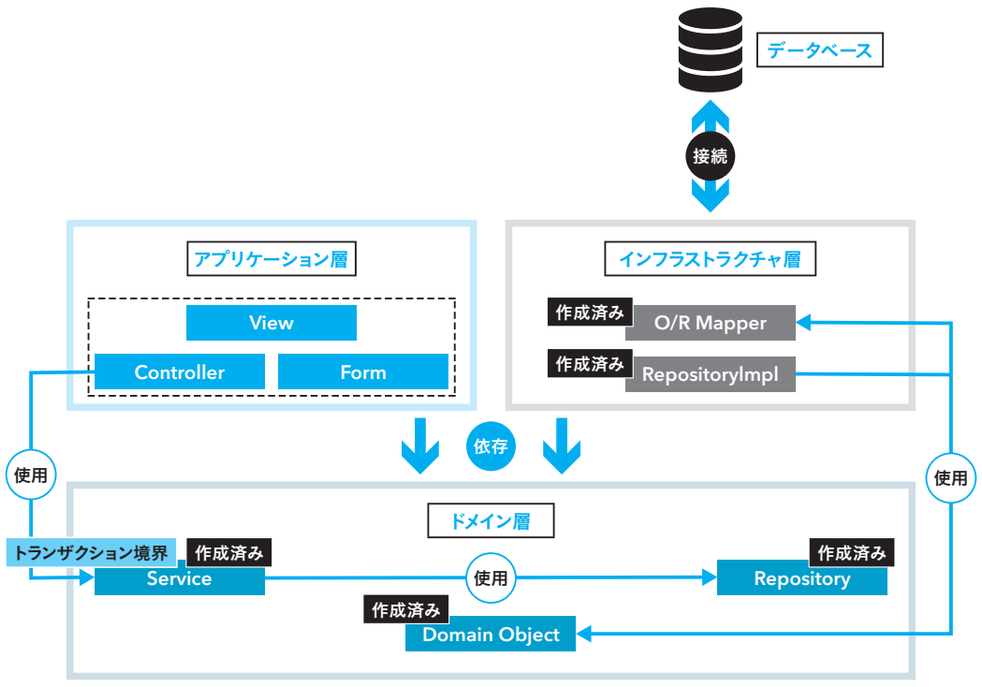


表12.1から作成名称を確認します。**表12.1**の中のNo.1～No.3が該当箇所になります。

表12.1 作成コンポーネント一覧

No	層	コンポーネント	名称	備考
1	アプリケーション層	View	-	見た目、画面です
2	アプリケーション層	Controller	QuizController	制御の役割を担います
3	アプリケーション層	Form	QuizForm	「画面のフォーム」を表現します
4	ドメイン層	Service	QuizService	インターフェースで作成します
5	ドメイン層	ServiceImpl	QuizServiceImpl	「Service」を実装します
6	ドメイン層	Domain Object	Quiz	ここでは「Entity」と同様です
7	ドメイン層	Repository	QuizRepository	インターフェースで作成します
8	インフラストラクチャ層	RepositoryImpl	-	O/Rマッパーにより自動作成されます
9	インフラストラクチャ層	O/R Mapper	-	Spring Data JDBCを使用します

作成する「htmlファイル」は3つです。機能は登録、変更、削除、参照、ゲームと5機能あるのに画面が何故3つ？と思った方もいるかもしれませんが、**図12.2**のように画面を作成します。

図12.2 作成画面ファイル

今回は「登録／変更／削除／参照」を「crud.html」の1ファイルで行います。「登録」と「変更」はそもそも「入力項目」が同じなので1つにまとめることができます。違いと言えば「登録」画面は画面表示時に「入力項目」の初期入力値が「空白」であり、「変更」画面は画面表示時に「入力項目」の初期入力値が「登録されているデータ」になります。

参照にあたる一覧表示ですが、登録されているクイズ情報データがあれば画面下部に「クイズ情報一覧」を表示し、クイズ情報が登録されていない場合、画面下部に「登録されているクイズはありません。」と表示します。

「crud」画面は上部エリアに「登録or変更」、下部エリアに「クイズ情報一覧」を表示します（**図12.3**）。