

Spring MVC

Ver.1.1

概要

- Spring MVC とは
- Spring MVC の開発環境

フレームワークとは開発効率や品質を向上させるための、便利な部品をまとめて提供してくれるもの。

アプリケーションをゼロから組み立てるのではなく、フレームワークを利用する開発が一般的。

- ・ Java言語で利用できるフレームワークの例

- Apache Struts

- JSF (Java Server Faces)

- Spring framework

- Play framework

- Seaser2

- 長所

多くの部品が提供されており、煩雑な処理をゼロから構築する必要がない。

決められた開発パターン(作法)で構築することになるため、高い保守性が期待できる。

フレームワークに当てはめて開発することで一定の品質が期待できる。

- 短所

特有の開発パターン(作法)を学習しなければならない。

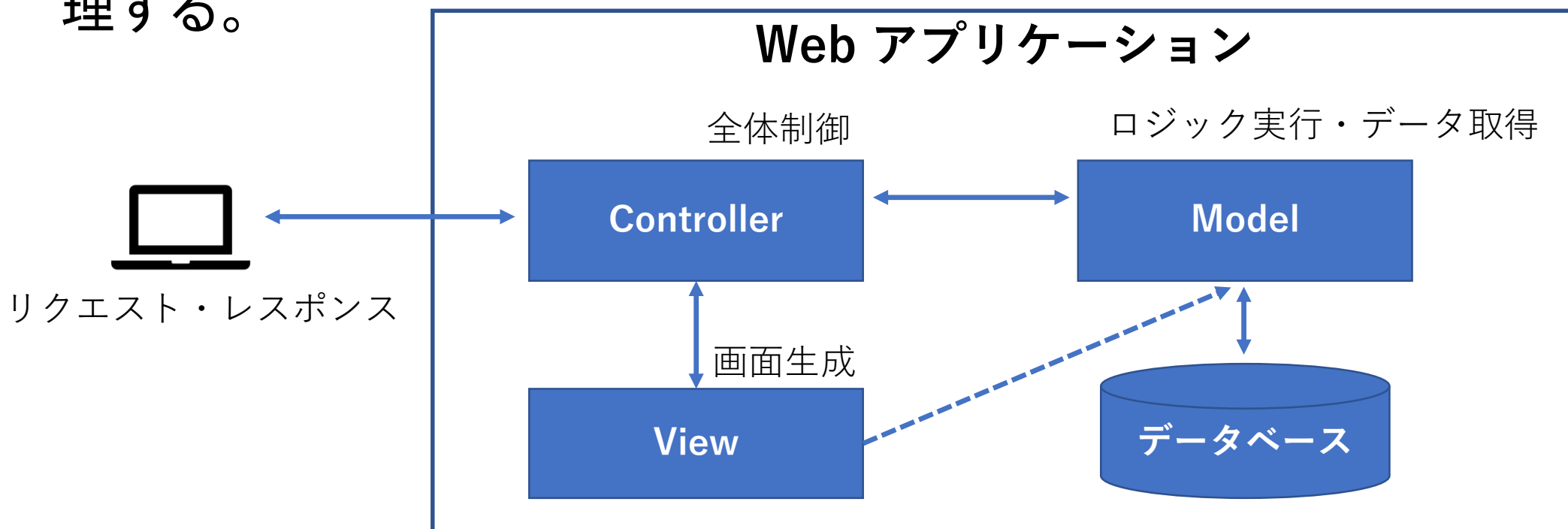
フレームワークが対応していない機能は、構築が(逆に)難しい場合がある。

Spring framework は世界中で利用されているフレームワーク。
DI(依存性注入)や AOP(アスペクト指向プログラミング)を支援するフレームワークとして有名。

Spring framework は 2004年にバージョン1.0がリリースされて以来、機能追加を繰り返し、2013年にバージョン4.0がリリースされている。サブプロジェクトが多数存在し、大きなシステムを形成している。

- Spring Core
- Spring MVC
- Spring Mobile
- Spring Web Service
- Spring Security
- Spring Data
- Spring Batch
- Spring Integration
- Spring Cloud
- Spring Tool Suite
- Spring IO Platform
- Spring Boot

Spring MVC は Web アプリケーションを開発する際に使用するフレームワークの 1 つ。
アーキテクチャとして MVC パターンを採用している。
アプリケーションを Model、View、Controller という 3つの役割のコンポーネントに分割して、クライアントからのリクエストを処理する。



Spring Tool Suite (Spring の開発環境)

- <http://spring.io/tools/sts/all>

日本語化パッチ

- <http://mergedoc.osdn.jp/>
 - Pleiades (プレアデス) → 最新版ダウンロード
 - ※ Readme を確認の上インストールすること

- ・ 前頁の Spring Tool Suite (STS) をインストールしましょう。
 - ・ インストールした STS に日本語化パッチをあてましょう。
- ※ インストール媒体は講師に確認のこと。

以下の手順で Spring MVC の新規プロジェクトを作成する。

- ・ [ファイル] → [新規] → [Spring レガシー・プロジェクト]
→ [Spring MVC Project] を選択。
- ・ プロジェクト名 : LessonSpringMVC
- ・ パッケージ : jp_co.good_works.lesson

Maven (※) を実行し必要なライブラリをダウンロードする。

- LessonSpringMVC を右クリック
→ [Maven] → [プロジェクトの更新] を実行
- LessonSpringMVC を右クリック
→ [実行] → [Maven install] を実行

※ Maven: プロジェクトのライフサイクル管理ツール

ここでは、Spring MVC が依存する(必要とする)ライブラリの取得に利用している。

- ・「サーバ」ビューの Pivotal tc Server Developer Edition を
右クリック
 - [追加および除去]
 - 「LessonSpringMVC」を追加して [完了] をクリック。
- ・「サーバ」ビューの「サーバを起動」ボタンをクリック。

前頁までの手順を実行したのち、以下の URL にアクセスして Hello world! 画面が表示されることを確認しましょう。

<http://localhost:8080/lesson/>

※ STS 以外に Eclipse が起動していたら停止してください。

・コントローラの確認

src/main/java/
jp.co.goodl_works.lesson
HomeController.java

コントローラクラスには @Controller アノテーションをつける。リクエストURLとの関連付けに @RequestMapping アノテーションをつける。
戻り (return) で使用するJSPを指定する。
※ home.jsp を使用する場合は “home” を戻す。

@Controller

```
public class HomeController {  
    <<省略>>  
    @RequestMapping(value = "/", method = RequestMethod.GET)  
    public String home(Locale locale, Model model) {  
        Date date = new Date();  
        DateFormat dateFormat = <<省略>>  
        String formattedDate = dateFormat.format(date);  
        model.addAttribute("serverTime", formattedDate );  
        return "home";  
    }  
}
```

サーバの現在時刻をJSPに受け渡す

- JSP の確認

src/webapp/WEB-INF/views/
home.jsp

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ page session="false" %>
<html>
<head>
<title>Home</title>
</head>
<body>
<h1>
Hello world!
</h1>

<P> The time on the server is ${serverTime}. </P>
</body>
</html>
```

コントローラでセットした情報（サーバの時刻）を表示

設定ファイルの確認

- web.xml (src/webapp/WEB-INF/web.xml)

Spring MVC を利用するうえでの決めりの設定。ただし、日本語(マルチバイト文字列)を正常に処理するためには文字コードに関する Filter の設定追加が必要(後述)。

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app ...>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/spring/root-context.xml</param-value>
  </context-param>
  <listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>
  <servlet>
    <servlet-name>appServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/spring/appServlet/servlet-context.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>appServlet</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
</web-app>
```

全ての Servlet/Filter に適用される共通設定ファイル(中身は空白)

Spring MVC の設定ファイル(次頁参照)

設定ファイルの確認

- `servlet-context.xml` (`src/webapp/WEB-INF/spring/appServlet/servlet-context.xml`)

Spring MVC の設定ファイル。Spring MVC の機能を追加したり、動作を変更する場合に編集が必要(後述)。

アノテーションによる制御を有効にする

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

```
<annotation-driven />
```

リソースファイル (HTML, JS, CSS 等) の置き場所

```
<resources mapping="/resources/**" location="/resources/" />
```

```
<beans:bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  <beans:property name="prefix" value="/WEB-INF/views/" />
  <beans:property name="suffix" value=".jsp" />
</beans:bean>
```

JSP ファイルの置き場所

```
<context:component-scan base-package="jp.co.good_works.lesson" />
```

```
</beans:beans>
```

ここで指定したパッケージ配下が Spring MVC の制御対象となる

コントローラとJSP

- リクエストマッピング
- Hello, Spring! の作成

リクエストマッピング ①

@RequestMapping アノテーションによって、リクエストURLにコントローラのメソッドを関連付ける(マッピングする)ことができる。

```
@RequestMapping(value = "/", method = RequestMethod.GET)
public String home(Locale locale, Model model) {
    <<省略>>
}
```

ここでは / に対する GET リクエストと、メソッド home を関連付けている。ここでのパス (/) は、アプリケーションルートパス (/lesson) 配下のパスを表している。

home メソッドが処理するURL:

<http://localhost:8080/lesson/>

リクエストマッピング ②

@RequestMapping アノテーションの例

```
@RequestMapping(value = "/hello", method = RequestMethod.GET)
```

→ http://localhost:8080/lesson/hello に対する GET リクエストと関連付く

```
@RequestMapping(value = "/hello", method = RequestMethod.POST)
```

→ http://localhost:8080/lesson/hello に対する POST リクエストと関連付く

```
@RequestMapping(value = "/hello")
```

→ http://localhost:8080/lesson/hello に対する リクエスト (全般) と関連付く

```
@RequestMapping(value = "/hello", params = "send")
```

→ http://localhost:8080/lesson/hello に対して、パラメタ (send) が送信された場合に場合に関連付く

リクエストマッピングしたメソッドの引数には、必要なオブジェクトを指定する(必須の引数が決められているわけではない)。

以下の例では、hello() でも、hello(Locale locale) でも、hello(Model model) でも、hello(HttpServletRequest request, Model model) でも動作する。

```
@RequestMapping(value = "/hello", method = RequestMethod.GET)
public String hello(Model model) {
    <<省略>>
}
```

必須の引数が決められているわけではない
必要なオブジェクトを指定する

リクエストマッピングしたメソッドの引数として指定できる主なオブジェクト。

- HttpServletRequest
- HttpSession
- Locale
- InputStream / Reader (リクエスト読み込み用)
- OutputStream / Writer (レスポンス書き込み用)
- Map
- RedirectAttributes
- **Model**
- **JavaBeans (Form オブジェクト)**
- **BindingResult**

リクエストマッピングしたメソッドの戻り値(文字列)が、連携する JSP の名前(拡張子は含めない)として認識される。

hello.jsp (src/webapp/WEB-INF/views/hello.jsp) と連携する場合の例:

```
@RequestMapping(value = "/hello", method = RequestMethod.GET)
public String hello(Model model) {
    model.addAttribute("message", "Hello, Spring!");
    return "hello";
}
```

JSP に情報を渡すには Model を利用する。
Model にセット(setAttribute)した情報が JSP で取り出せる。

JSP で表示(画面)系処理を構築する。

コントローラでセットした情報を取り出すことができる。

- ・ hello.jsp (src/webapp/WEB-INF/views/hello.jsp)

```
<html>
<head>
  <title>Hello</title>
</head>
<body>
  <p>${message}</p>
</body>
</html>
```

- ・ `http://localhost:8080/lesson/hello` にアクセスすると Hello, Spring! と画面上に表示するプログラムを作成しましょう。
 - コントローラ (`HelloController.java`) を `jp.co.good_works.lesson.springmvc.controller` パッケージに作成し、リクエストマッピングしたメソッドを作成。
 - ※ 作成したメソッドで Hello, Spring! という文字列を Model にセットする。
 - JSP (`hello.jsp`) を `WEB-INF/view/` に作成
 - コントローラでセットした文字列を画面に表示する
- ・ ビルトインで作成された `HomeController.java` を `jp.co.good_works.lesson.springmvc.controller` パッケージに移しましょう。

コントローラで以下の文字列をセットすると、JSP で正常に画面が表示されない。これは `<script>` が HTML のタグとして解釈され、JavaScript の `alert()` 文が実行されるため。

```
@RequestMapping(value = "/hello", method = RequestMethod.GET)
public String hello(Model model) {
    model.addAttribute("message", "<script>alert()</script>");
    return "hello";
}
```

HTML エンコードを施すことで、HTML のタグも画面上に表示することができる。

上記 (`<script>alert()</script>`) を HTML エンコードした文字列:

`<script>alert()</script>`

JSP で標準タグライブラリ(<c:out>)を使用することで、HTML エンコードを施した文字列を表示することができる。

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
    <title>Hello</title>
</head>
<body>
    <p><c:out value="$ {message}" /></p>
</body>
</html>
```

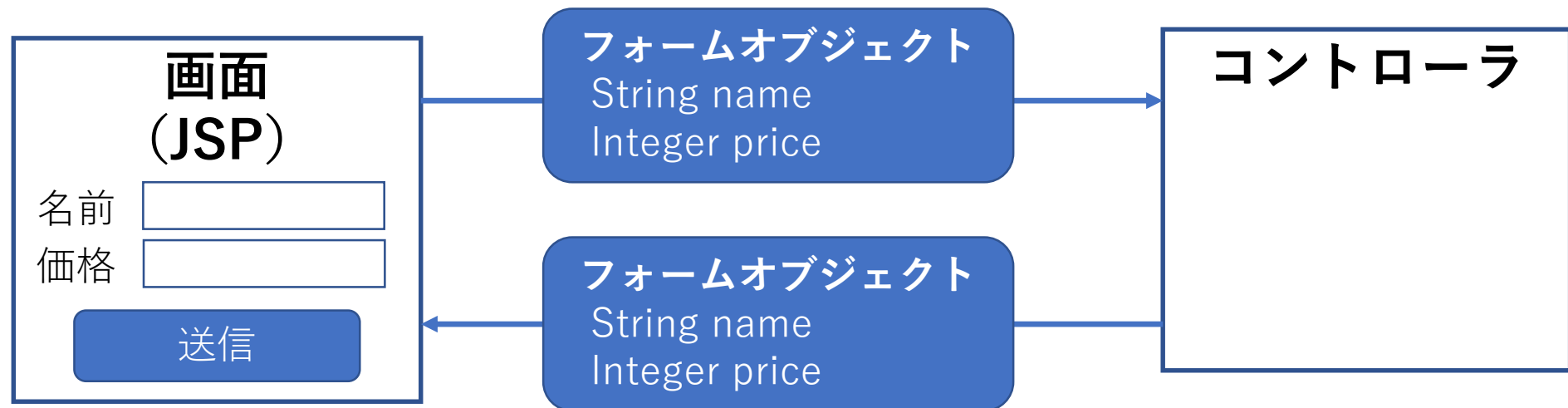
- ・ hello.jsp に前頁のHTMLエンコードを施しましょう。
- ・ <script> タグが画面上に正しく表示されることを確認しましょう。

フォーム

- ・ フォームによるパラメタの送信
- ・ 商品情報送信アプリケーションの作成

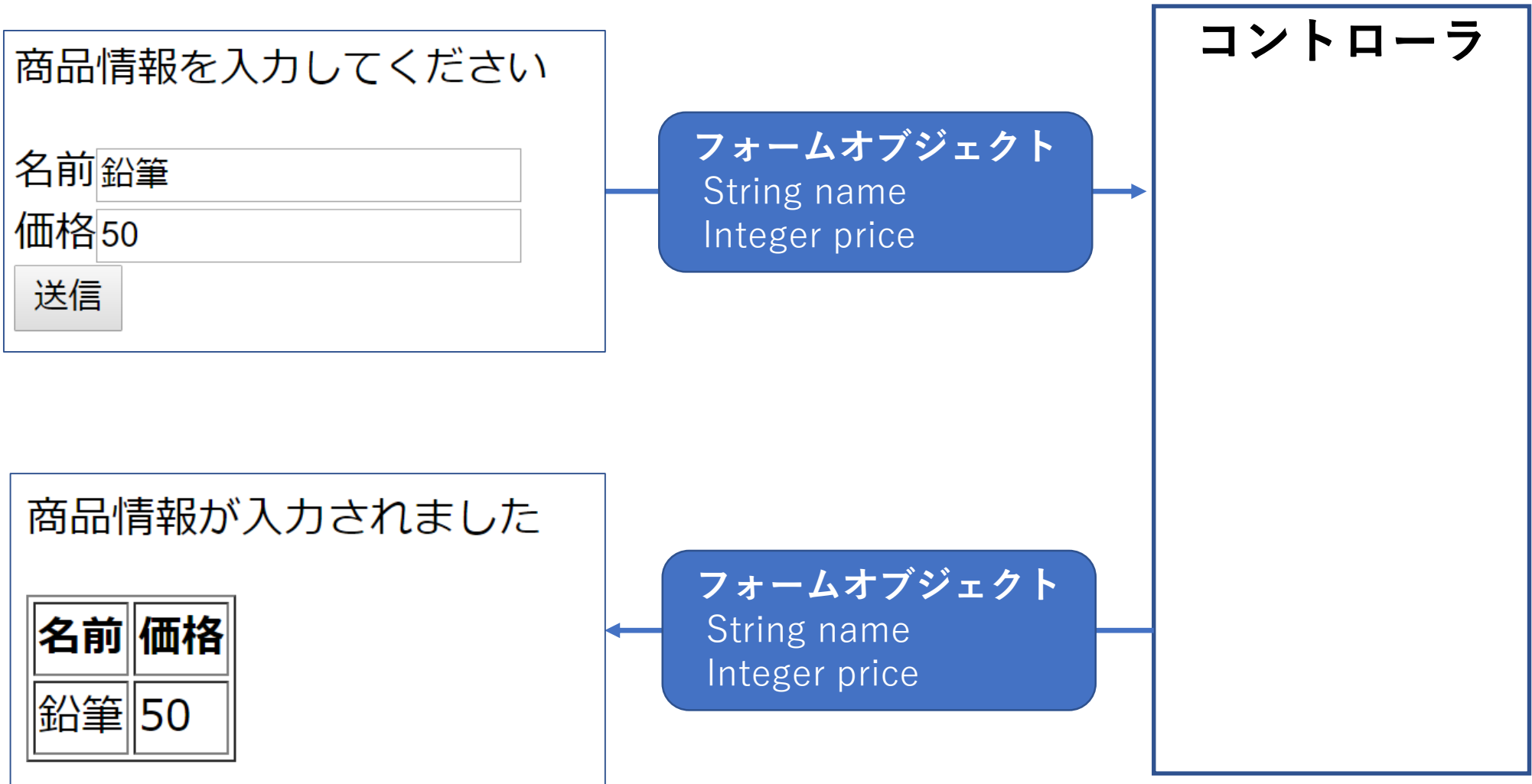
フォームによるパラメタの送信

フォームによるパラメタの送信は、ユーザから情報を受け取る処理の基本。Spring MVC ではリクエストやレスポンスを直接操作せず、フォームの値を管理するフォームオブジェクト（フォームクラス）を利用する。



商品情報送信プログラム

例題として、商品情報送信プログラムを作成する。



商品情報を管理するフォーム(ProductForm) クラスを作成する。

```
package jp_co.good_works.lesson.springmvc.form;
public class ProductForm {
    private String name;
    private Integer price;
    public void setName(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void setPrice(Integer price) {
        this.price = price;
    }
    public Integer getPrice() {
        return price;
    }
}
```

商品情報入力 JSP の作成

商品情報を入力する画面(JSP)を作成する [WEB-INF/view/product.jsp]

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<html>
<head>
    <meta charset="utf-8">
    <title>Product</title>
</head>
<body>
<p><c:out value="${message}" /></p>
<form:form modelAttribute="productForm">
    名前<form:input path="name" /><br/>
    価格<form:input path="price" /><br/>
    <input type="submit" value="送信"><br/>
</form:form>
</body>
</html>
```


フォームオブジェクトの名前（先頭を小文字にする）

フォームオブジェクトのプロパティ名

商品情報出力 JSP の作成

入力された商品情報の出力画面を作成する [WEB-INF/view/product_result.jsp]

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<html>
<head>
    <meta charset="utf-8">
    <title>Product Result</title>
</head>
<body>
<p><c:out value="${message}" /></p>
<table border="1">
    <tr><th>名前</th><th>価格</th></tr>
    <tr>
        <td><c:out value="${productForm.name}" /></td><td>${productForm.price}</td>
    </tr>
</table>
</body>
</html>
```



フォームオブジェクトの名前, プロパティ名

商品情報コントローラの作成

```
package jp_co.good_works.lesson.springmvc.controller;
```

```
<<import 文は省略>>
```

```
@Controller
```

```
public class ProductController {
```

```
    @RequestMapping(value = "/product", method = RequestMethod.GET)
```

```
    public String product(Model model) {
```

```
        ProductForm form = new ProductForm();
```

```
        model.addAttribute("message", "商品情報を入力してください");
```

```
        model.addAttribute("productForm", form);
```

```
        return "product";
```

```
    }
```

```
    @RequestMapping(value = "/product", method = RequestMethod.POST)
```

```
    public String product(Model model, @ModelAttribute ProductForm form) {
```

```
        model.addAttribute("message", "商品情報が入力されました");
```

```
        model.addAttribute("productForm", form);
```

```
        return "product_result";
```

```
    }
```

```
}
```

ブラウザにURLを指定してアクセスしたら
呼ばれるメソッド

JSP にフォーム
を受け渡す

入力用JSP

送信ボタンをクリックしたら呼ばれるメソッド

JSP にフォー
ムを受け渡す

出力用JSP

JSPから受け取るフォームオブジェクトには
@ModelAttribute をつける。
このフォームオブジェクトに画面で入力され
た情報が格納されている。

日本語(マルチバイト文字列)を正常に処理するためには、web.xmlに文字コードに関する Filter の設定追加が必要 [WEB-INF/web.xml]

```
...
<filter>
  <filter-name>CharacterEncoding</filter-name>
  <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
  <init-param>
    <param-name>encoding</param-name>
    <param-value>UTF-8</param-value>
  </init-param>
  <init-param>
    <param-name>forceEncoding</param-name>
    <param-value>true</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>CharacterEncoding</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
...
```

商品情報送信のプログラムを作成しましょう。

- src/main/java
 - jp_co/good_works/lesson/springmvc/form/ProductForm.java
 - jp_co/good_works/lesson/springmvc/controller/ProductController.java
- src/main/webapp/WEB-INF/views/
 - product.jsp（入力画面）
 - product_result.jsp（出力画面）
- src/main/webapp/WEB-INF/
 - web.xml（文字化け対策）

前頁の商品情報送信プログラムについて、以下の機能追加をしましょう。

- ・ 名前欄の初期状態を「名前を入力してください」にします。
- ・ 価格欄の初期状態を「0」にします。
- ・ 名前もしくは価格が入力されずに送信ボタンが押されたら、再度商品情報入力画面を表示します。

コントローラのメソッドには `HttpServletRequest` を指定することもできるので、(いざとなれば) サーブレットと同じように `HttpServletRequest` を使って処理を構築することもできる。

```
...
@RequestMapping(value = "/product_req", method = RequestMethod.POST)
public String product(Model model, HttpServletRequest request) throws java.io.IOException {
    request.setCharacterEncoding("UTF-8");
    String name = request.getParameter("name");
    String price = request.getParameter("price");
    ProductForm form = new ProductForm();
    form.setName(name);
    form.setPrice(Integer.parseInt(price));
    request.setAttribute("message", "商品情報が入力されました (HttpServletRequest利用)");
    request.setAttribute("productForm", form);
    return "product_req_result";
}
...
```

ModelAndView は、ビューとモデルを扱えるようにしたもの。
必要な情報をひとまとめにしてやり取りすることができる。

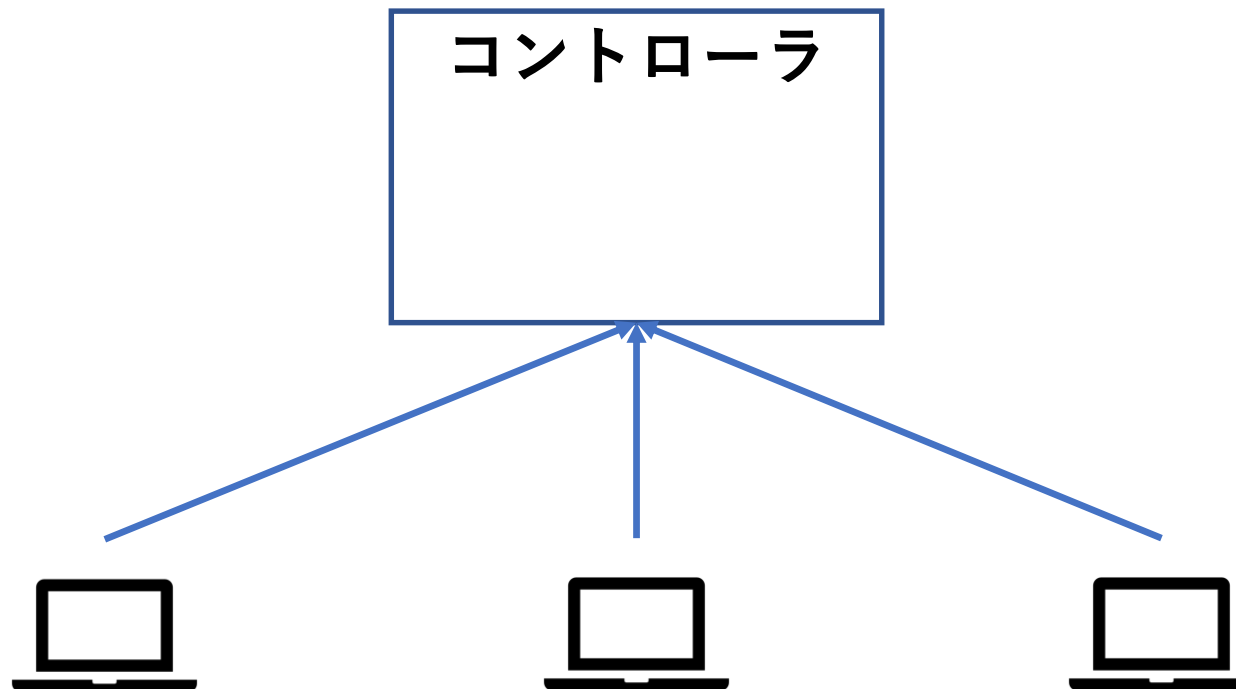
```
...  
@RequestMapping(value = "/product", method = RequestMethod.POST)  
public ModelAndView product(@ModelAttribute ProductForm form) {  
    ModelAndView modelAndView = new ModelAndView("product_result");  
    modelAndView.addObject("message", "商品情報が入力されました");  
    modelAndView.addObject("productForm", form);  
    return modelAndView;  
}  
...
```



セッション毎の情報管理

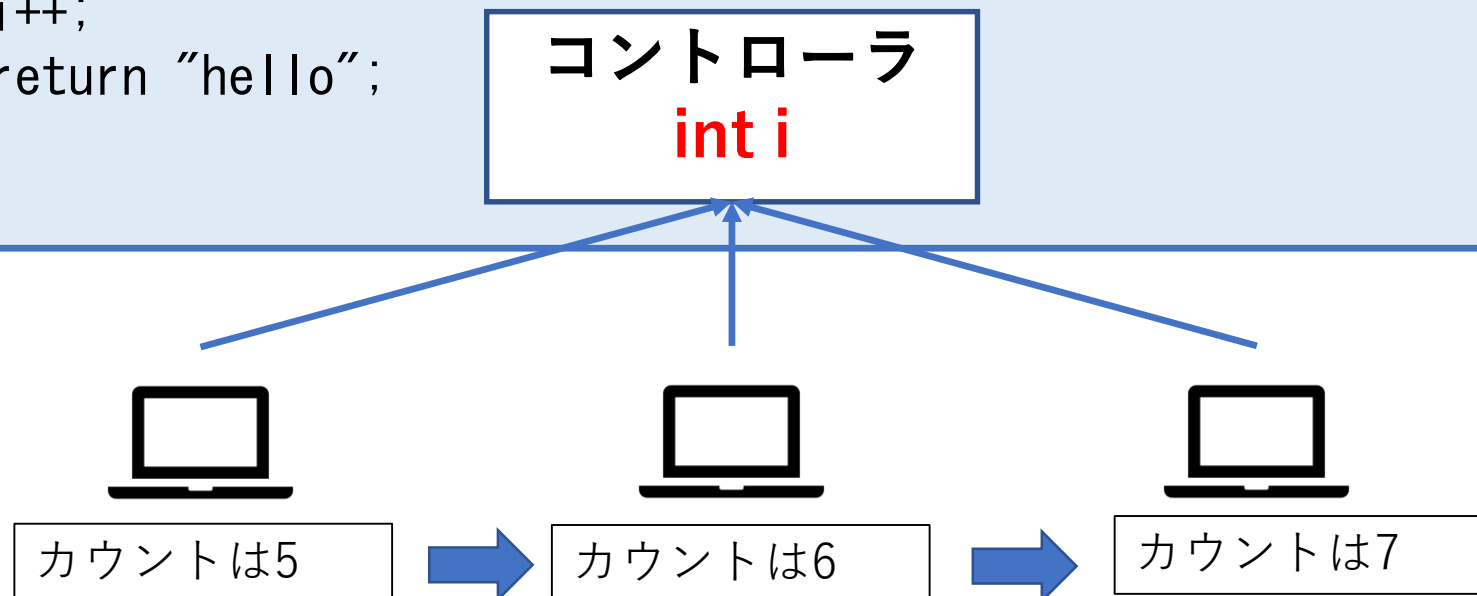
- ・ セッションスコープ
- ・ 商品情報のリスト管理

これまで作成してきたコントローラは、1つのインスタンスをすべてのユーザが共有している。



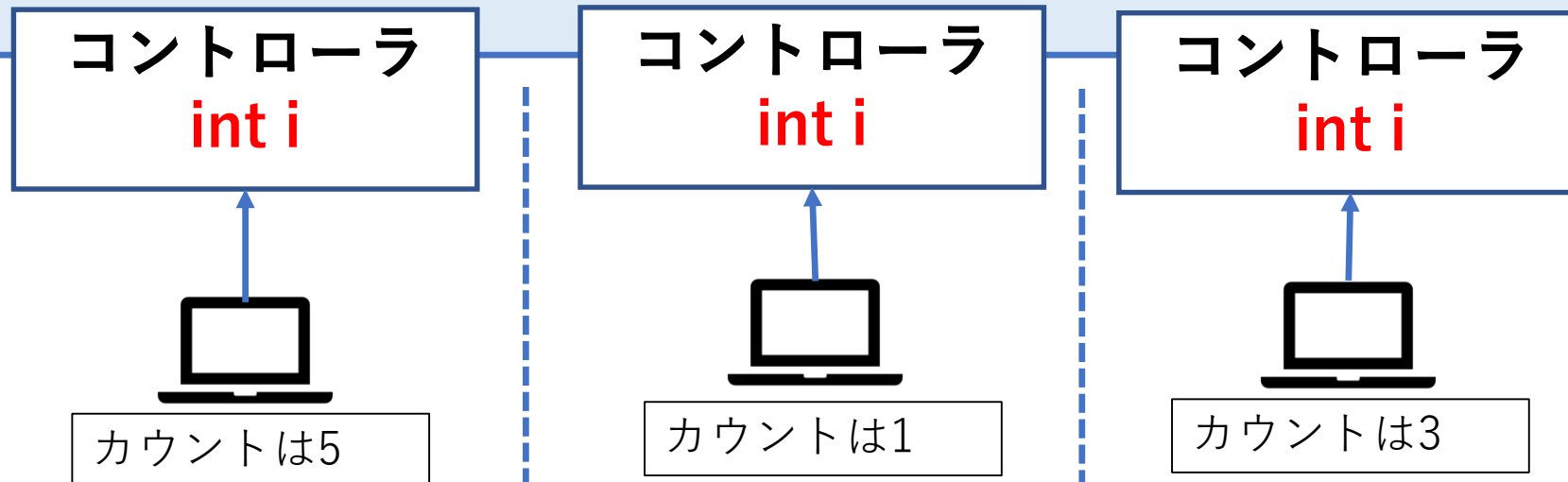
実験として、コントローラにインスタンスフィールド(i)を用意し、アクセスがある度にカウントアップしてみる。

```
...  
private int i = 0;  
  
@RequestMapping(value = "/hello", method = RequestMethod.GET)  
public String hello(Model model) {  
    model.addAttribute("message", "カウントは" + i);  
    i++;  
    return "hello";  
}  
...
```



コントローラのスコープをセッションスコープに変えてみる。コントローラのインスタンスフィールド (i) はセッション毎の情報として保持される。

```
import org.springframework.context.annotation.Scope;  
@Scope("session")  
@Controller  
public class HelloController {  
    private int i = 0;  
    <<省略>>  
}
```



Goodworks
technology for the enterprise

商品情報が入力されました

名前	万年筆
価格	1000
送信	

フォームオブジェクト

String name

Integer price

商品情報

コントローラ(ProductController)のスコープをセッションスコープとし、商品情報リストをインスタンス変数で保持する。商品情報が送信されるたびに商品情報リストに商品情報を追加する。

```
@Controller
```

```
@Scope("session")
```

```
public class ProductController {
```

```
    private List<ProductForm> productList = new ArrayList<ProductForm>();
```

```
    @RequestMapping(value = "/product", method = RequestMethod.POST)
```

```
    public String product(Model model, @ModelAttribute ProductForm form) {
```

```
        if (form.getName() == null || form.getPrice() == null) {
```

```
            model.addAttribute("message", "商品情報が空です");
```

```
        } else {
```

```
            model.addAttribute("message", "商品情報が入力されました");
```

```
            productList.add(form);
```

```
        }
```

```
        model.addAttribute("productForm", form);
```

```
        model.addAttribute("productList", productList);
```

```
        return "product";
```

インスタンスフィールドとして、商品情報リストを追加

商品情報リストに商品情報を追加

同じJSPで入力と出力を行う

JSPに商品情報リストを受け渡す

JSP (product.jsp) に商品情報リストの出力処理を追加する。

```
...  
<c:if test="${not empty productList}">  
  <table border="1">  
    <tr><th>名前</th><th>価格</th></tr>  
    <c:forEach var="productForm" items="${productList}">  
      <tr>  
        <td><c:out value="${productForm.name}"></c:out></td>  
        <td><c:out value="${productForm.price}"></c:out></td>  
      </tr>  
    </c:forEach>  
  </table>  
  <br>  
</c:if>  
...
```

If 文(商品情報リストが空でなければ)

フォームオブジェクトのプロパティ名

for 文(商品情報リストから 1 件ずつ商品情報を取り出し処理する)

商品情報をリストで管理するプログラムを作成しましょう。

- src/main/java
 - jp_co/good_works/lesson/springmvc/controller/ProductController.java
- src/main/webapp/WEB-INF/views/
 - product.jsp（入力・出力画面）

バリデーション

- 入力チェック
- エラーメッセージ

Spring framework には、入力された値をチェックする仕組み(バリデーション)があり、値が妥当でない場合にはエラーメッセージを表示できる。

エラーがあります

名前

may not be empty

価格 -10

must be greater than or equal to 10

送信

入力チェック機能を利用するには、ライブラリの追加が必要

- ・ 設定ファイル(pom.xml)に定義を追加する

```
<!-- add for validation -->
<dependency>
    <groupId>javax.validation</groupId>
    <artifactId>validation-api</artifactId>
    <version>1.1.0.Final</version>
</dependency>
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>5.0.1.Final</version>
</dependency>
```

- ・ LessonSpringMVC を右クリック

→ [実行] → [Maven install] を実行

フォームのインスタンスフィールドに、アノテーションによる各種制約を付与することができる。

- ・ 必須制約 (文字列: @NotEmpty, 数値: @NotNull)
- ・ 最大値、最小値制約 (@Max, @Min)
- ・ メール形式制約 (@Email)

※ ProductForm に制約を追加:

```
import javax.validation.constraints.*;
import org.hibernate.validator.constraints.*;
public class ProductForm {
    @NotEmpty
    private String name;

    @NotNull
    @Min(value = 10) @Max(value = 10000)
    private Integer price;
```

コントローラ(ProductController)にエラー処理を追加する。

```
import org.springframework.validation.BindingResult;
import org.springframework.validation.annotation.Validated;
<<省略>>
@RequestMapping(value = "/product", method = RequestMethod.POST)
public String product(Model model,
    @Validated @ModelAttribute ProductForm form, BindingResult result) {
    if (result.hasErrors()) {
        model.addAttribute("message", "エラーがあります");
    } else {
        model.addAttribute("message", "商品情報が入力されました");
        productList.add(form);
    }
    model.addAttribute("productForm", form);
    model.addAttribute("productList", productList);
    return "product";
}
```

バリデーション制御するフォームに @Validated を付与する

チェック結果を BindingResult で受け取れる。エラーがある場合、このオブジェクトの hasErrors() メソッドが true を返却する

エラーメッセージの出力

JSP (product.jsp) にエラーメッセージの出力を追加する。

```
...  
<form:form modelAttribute="productForm">  
  名前<form:input path="name" /> <form:errors path="name" cssStyle="color:red"/><br/>  
  価格<form:input path="price" /> <form:errors path="price" cssStyle="color:red"/><br/>  
  <input type="submit" value="送信"><br/>  
</form:form>  
...
```

設定ファイル(servlet-context.xml)にメッセージファイルの設定を追加する。

```
<beans:bean id="messageSource"  
class="org.springframework.context.support.ReloadableResourceBundleMessageSource">  
    <beans:property name="basename" value="classpath:ValidatorMessages" />  
</beans:bean>
```

以下の場所にメッセージファイルを用意する。

- src/main/resources/

ValidatorMessages_ja.properties

用意した ValidatorMessages_ja.properties に日本語メッセージを定義する。

- 定義形式

[アノテーション名] = [メッセージ]

typeMismatch.[型名] = [メッセージ]

typeMismatch. java. lang. Integer={0} は数値で指定してください。
NotEmpty={0} は必須です。
NotNull={0} は必須です。
Max={0} は {1} 以下の値を指定してください。
Min={0} は {1} 以上の値を指定してください。

{0} … プロパティ名に置き換わる

{1} 以降 … 制約値に置き換わる

※ Eclipse で編集するとユニコードエスケープされるが、これで問題ない。

typeMismatch. java. lang. Integer={0} ¥u306F¥u6570¥u5024¥u306…
NotEmpty={0} ¥u306F¥u5FC5¥u9808¥u3067¥u3059
NotNull={0} ¥u306F¥u5FC5¥u9808¥u3067¥u3059
Max={0} ¥u306F {1} ¥u4EE5¥u4E0B¥u306E¥u5024¥u3092¥u6307¥u5B9A…
Min={0} ¥u306F {1} ¥u4EE5¥u4E0A¥u306E¥u5024¥u3092¥u6307¥u5B9A…

エラーメッセージの日本語対応

実行結果:

エラーがあります

名前 nameは必須です

価格 priceは10,000以下の値を指定してください。

プロパティ名も日本語化する:

typeMismatch. java. lang. Integer={0} は数値で指定してください。

NotEmpty={0} は必須です。

NotNull={0} は必須です。

Max={0} は {1} 以下の値を指定してください。

Min={0} は {1} 以上の値を指定してください。

name=名前

price=価格

実行結果:

エラーがあります

名前 名前は必須です

価格 価格は10以上の値を指定してください。

商品情報に不正な値が入力された場合に、エラーメッセージ(日本語)を表示するプログラムを作成しましょう。

- pom.xml
- src/main/java
 - jp_co/good_works/lesson/springmvc/controller/ProductForm.java
 - jp_co/good_works/lesson/springmvc/controller/ProductController.java
- src/main/webapp/WEB-INF/views/
 - product.jsp
- src/main/resources/
 - ValidatorMessages_ja.properties

タグライブラリ

- セレクトボックス
- ラジオボタン
- チェックボックス
- テキストエリア

ユーザ情報送信のプログラムを作成しましょう。

ユーザ情報が入力されました

名前	年齢	性別	E-mail	出身地	好きな言語	備考
山田 太郎	27	男	yamada@dummy.co.jp	関東	[Java, Python]	趣味はランニング

名前：

年齢：

性別： ☒ 男 ☐ 女

E-mail：

出身地：

好きな言語： ☒ Java ☐ PHP ☐ Ruby ☐ Perl ☒ Python

備考：

コントローラ

ユーザリスト

ユーザ情報

ユーザ情報

フォームオブジェクト

ユーザ情報

- ・ 名前（テキストボックス）※必須、初期値：入力してください
- ・ 年齢（テキストボックス）※必須、数値（20以上）
- ・ 性別（ラジオボタン）※男・女から選択する
- ・ 出身地（セレクトボックス）
北海道・東北・関東・中部・近畿・中国・四国・九州, から選択する（複数選択不可）
- ・ 好きな言語（チェックボックス）
Java・C#・C/C++・PHP・Perl・Ruby・Python, から選択する（複数選択可）
- ・ 備考（テキストエリア）

ユーザ情報を入力してください

名前：

年齢：

性別： ☐ 男 ☐ 女

E-mail：

出身地：

好きな言語： ☐ Java ☐ PHP ☐ Ruby ☐ Perl ☐ Python

備考：

入力されたユーザ情報をリストで出力します。

入力と出力を1つの JSP (userinfo.jsp) で構築します。

- src/main/java
 - jp_co/good_works/lesson/springmvc/form/UserInfoForm.java
 - jp_co/good_works/lesson/springmvc/controller/UserInfoController.java
- src/main/webapp/WEB-INF/views/
 - userinfo.jsp
- src/main/resources/
 - ValidatorMessages_ja.properties

- Form

```
private String gender;  
※ アクセサメソッドを追加すること
```

- JSP

プロパティ名

```
<form:radiobuttons path="gender" items="${genders}" />
```

- Controller

選択候補

```
...  
List<String> genders = new ArrayList<String>();  
genders.add("男");  
genders.add("女");  
model.addAttribute("genders", genders);  
...
```

・ Form

```
private String birthplace;  
※ アクセサメソッドを追加すること
```

・ JSP

プロパティ名

複数選択不可

```
<form:select path="birthplace" items="${birthplaces}" multiple="false" />
```

・ Controller

選択候補

```
...  
List<String> birthplaces = new ArrayList<String>();  
birthplaces.add("");  
birthplaces.add("北海道");  
birthplaces.add("東北");  
...  
model.addAttribute("birthplaces", birthplaces);  
...
```

- Form

```
private List<String> favoriteLangs;
```

※ アクセサメソッドを追加すること

- JSP

プロパティ名

```
<form:checkboxes path="favoriteLangs" items="${langs}" />
```

- Controller

選択候補

```
...  
List<String> langs = new ArrayList<String>();  
langs.add("Java");  
langs.add("PHP");  
langs.add("Ruby");  
...  
model.addAttribute("langs", langs);  
...
```

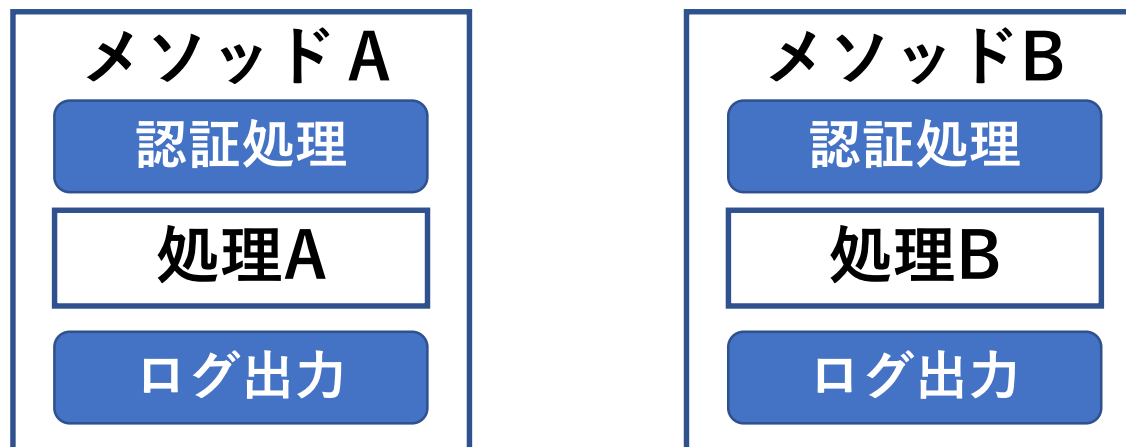

AOPと認証処理

- AOP
- 認証処理

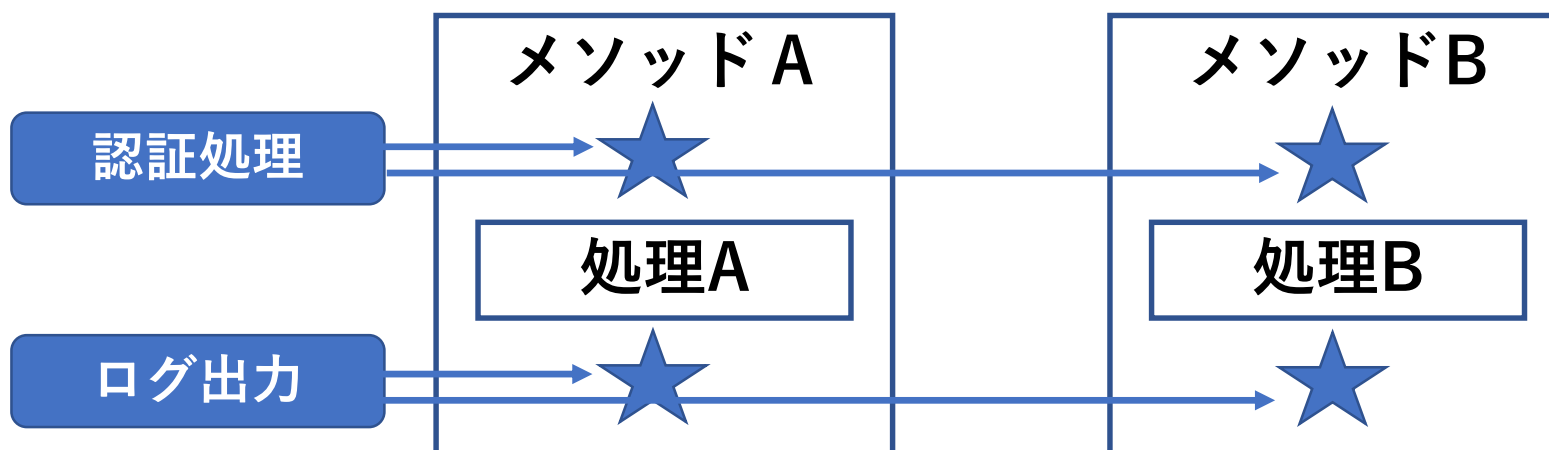
Spring framework は AOP (Aspect Oriented Programming: アスペクト指向プログラミング) を支援するフレームワーク。

AOP では、ログの出力や、トランザクション管理、認証処理等の共通的な処理(横断的関心事)を元のソースコードを変更せずに追加することができる。

通常のプログラミング:



AOP では元のソースコードを変更せずに(手をいれずに)、横断的関心事を織り込むことができる。



Spring AOP を利用するには、ライブラリの追加が必要

- ・ 設定ファイル(pom.xml)に定義を追加する

```
<!-- add for validation -->
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.8.10</version>
</dependency>
<dependency>
    <groupId>cglib</groupId>
    <artifactId>cglib</artifactId>
    <version>2.2</version>
</dependency>
```

- ・ LessonSpringMVC を右クリック
→ [実行] → [Maven install] を実行

Spring AOP を利用するには、
設定ファイル (servlet-context.xml) に以下を追加します。

```
xmlns:aop="http://www.springframework.org/schema/aop"  
...  
xsi:schemaLocation="  
...  
http://www.springframework.org/schema/aop  
http://www.springframework.org/schema/aop/spring-aop.xsd  
">  
...  
<aop:aspectj-autoproxy />  
...
```

例題として、認証処理（ログイン処理）を作成し、先に作成した HelloController に適用します。

- LoginForm

```
@NotEmpty  
private String userId;
```

```
@NotEmpty  
private String password;
```

※アクセサメソッドを追加すること

・ LoginController

```
package jp_co.good_works.lesson.springmvc.controller.no_cert;  
@Controller  
@Scope("session")  
public class LoginController {  
  
    private LoginLogic loginLogic = new LoginLogic();  
  
    public boolean isLive() {  
        return loginLogic.isLive();  
    }  
  
    @RequestMapping(value = "/login", method = RequestMethod.GET)  
    public String initializeLogin(Model model) {  
        ...  
    }  
}
```

<<次頁に続く>>

・ LoginController

<<前頁からの続き>>

```
@RequestMapping(value = "/login", method = RequestMethod.POST)
public String executeLogin(Model model,
    @Validated @ModelAttribute LoginForm form, BindingResult result,
    RedirectAttributes redirectAttr) {
    if (!result.hasErrors()) {
        try {
            loginLogic.login(form.getUserId(), form.getPassword());
            return "redirect:/hello";
        } catch (LoginException ex) {
            model.addAttribute("message", ex.getLocalizedMessage());
        }
    } else {
        model.addAttribute("message", "ユーザID、パスワードを入力してください。");
    }
    return "login";
}
```

・ LoginLogic

```
package jp_co.good_works.lesson.springmvc.logic;
public class LoginLogic {
    private LoginInfo loginInfo = null; // LoginForm とは別に作成
    /* ログインを実行する */
    public LoginInfo login(String userId, String password)
                                   throws LoginException {
        // 認証ロジックは簡易的でよい。
        // ※ guest/guestguest の場合は認証OKとする。
        // 認証OKの場合、変数 loginInfo をインスタンス化し認証情報
        // を格納して返却する。
        // 認証NGの場合、例外を発行する。
        throw new LoginException("ユーザIDまたはパスワードが違います。");
    }
    /* ログイン済かどうかを確認する */
    public boolean isLive() {
        // loginInfo がインスタンス化されていれば true を返却する
    }
}
```

- ・ LoginInfo (jp_co. good_works. lesson. springmvc. logic パッケージ)

```
@NotEmpty  
private String userId;
```

```
@NotEmpty  
private String password;
```

※アクセサメソッドを追加すること

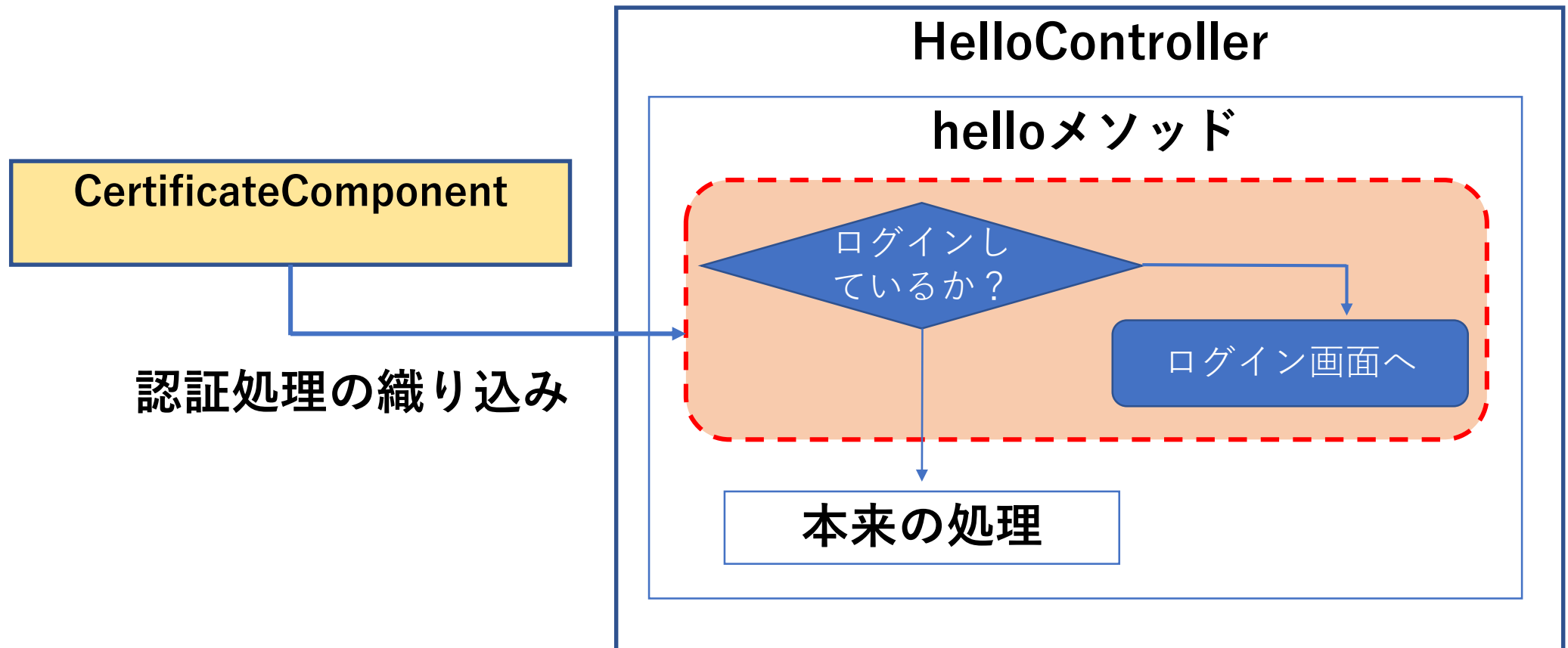
- LoginException (jp_co.good_works.lesson.springmvc.exception)

```
package jp_co.good_works.lesson.springmvc.exception;  
  
public class LoginException extends Exception {  
  
    public LoginException(String message) {  
        super(message);  
    }  
}
```

- login.jsp (WEB-INF/views/login.jsp)

```
<!DOCTYPE html>
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<html>
  <head>
    <meta charset="utf-8">
    <title>Welcome</title>
  </head>
  <body>
    <h2>${message}</h2>
    <form:form modelAttribute="loginForm">
      <form:input path="userId" />&nbsp;
      <form:errors path="userId" cssStyle="color:red"/><br>
      <form:password path="password" />&nbsp;
      <form:errors path="password" cssStyle="color:red"/><br>
      <input type="submit">
    </form:form>
  </body>
</html>
```

作成した認証処理を AOP により、HelloController クラスに織り込む。



▪ CertificateComponent

```
package jp_co.good_works.lesson.springmvc.aop;  
import ...  
import org.springframework.web.servlet.support.WebContentGenerator;  
@Aspect  
@Component  
public class CertificateComponent extends WebContentGenerator {  
  
    @Around("execution(* jp_co.good_works.lesson.springmvc.controller.HelloController.*(..))")  
    public String checkAuthenticated(ProceedingJoinPoint joinPoint) throws Throwable {  
        LoginController loginControl = getApplicationContext().getBean(LoginController.class);  
        if (!loginControl.isLive()) {  
            return (String) joinPoint.proceed();  
        }  
        return "redirect:/login";  
    }  
}
```

← 本来の処理

- ・ これまで作成した全てのコントローラ（LoginController を除く）、
に認証処理を追加しましょう。

- Dao を作成し、認証処理をデータベースにアクセスして実現しましょう。
 - src/main/java
 - jp_co/good_works/lesson/springmvc/dao/UserDao.java
- メソッド: `public UserInfo select(String userId, String password)`
- 該当するユーザがデータベースに存在すれば、該当ユーザの `UserInfo` を返却する。
- 該当するユーザがデータベースに存在しなければ、`null` を返却する。