

A faint, light blue world map is visible in the background of the slide, centered behind the text.

INTERNET ACADEMY

Institute of Web Design & Software Services

Spring Boot 2

インターネット・アカデミー

Spring Boot 2 目次

- Spring MVCアーキテクチャ概要
- 基礎編 1 (Controller + View)
- 基礎編 2 (Controller + View + Model)

A faint, light blue world map is visible in the background of the slide, centered behind the title text.

Spring MVCアーキテクチャー概要

Spring MVCアーキテクチャー概要

- Spring MVCアーキテクチャーとは
- Dispatcher Servlet
- DI（依存性注入）

Spring MVCアーキテクチャとは

Controller

アクセスされたURLに応じて
適切なModel, Viewを呼び出す

コントローラークラス(~.java)

View

表示画面
を定義

Thymeleaf(~.html)

Model

- ・ 1つ1つの処理
- ・ データの保持

モデルクラス(~.java)

Spring MVCアーキテクチャとは

クライアントがアクセスするURLに対応するModelを呼び出して、ViewのHTMLを出力させるよう、処理全体の管理を行う

Webクライアント
(ブラウザ)



Controller



Model



田中

name



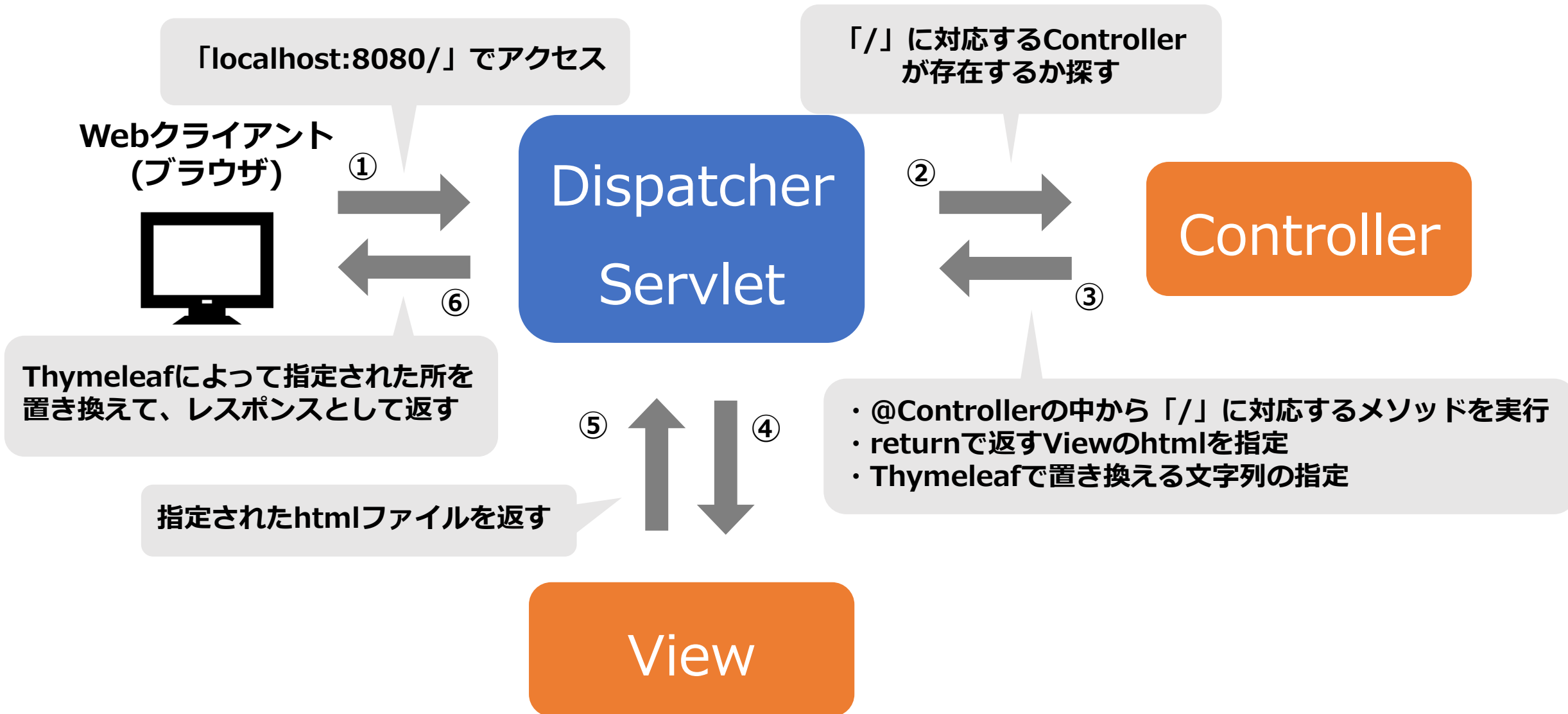
View



データベース操作や送付データの保持など、1つ1つの処理を担当する

クライアントに返すHTMLを担当する

【補足】 Dispatcher Servlet



DIとは

Dependency Injection(依存性注入)

直接インスタンス(依存性)を生成(new)せず、あらかじめ作って待機させておき、必要になったときに取り込む(注入)こと。

オブジェクトを再利用するため、処理スピードを速めることができる



A faint, light blue world map is visible in the background of the slide, centered behind the title text.

基礎編 1 (Controller + View)

Spring MVCアーキテクチャー概要

- Controllerの書き方
- Thymeleafを用いたViewの書き方
- 実践：お問い合わせフォームの入力画面の作成（C + V）

Controllerの書き方

1. Controllerのクラスであることを示す

FormController.java

@Controller

```
public class FormController {  
  
    . . .  
  
}
```

※補足

@Controllerの代わりに「@RestController」と記述すると、レスポンスがHTMLでなくJSONやXMLになり、Web API(REST-API)の開発ができる。

Controllerの書き方

2. クライアントからのリクエストURLに対応する処理（マッピング）を行う

FormController.java

```
@Controller
public class FormController {
    @RequestMapping("/sample")
    public String sample() {
        return "index";
    }
}
```

localhost:8080/sample(ドメイン以下が「/sample」のとき)
でアクセスされたとき呼び出される

戻り値が指定するものは「htmlファイル」であり、index→index.htmlを表示となる
(そのため、戻り値型がString型となっている)

※補足

@RequestMappingは以下のように属性の指定が可能

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

リクエストURL GET や POST などのメソッド指定

Controllerの書き方

3. index.htmlにデータを渡す

FormController.java

```
@Controller
```

```
public class FormController {
```

```
    @RequestMapping("/sample")
```

〇〇.addAttribute(・・・); の〇〇と一致させる

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

```
        model.addAttribute("message", "Hello World");
```

```
        return "index";
```

Thymeleafでの「message」を「Hello World」に置き換える

```
    }
```

```
}
```

Thymeleafを用いたViewの書き方

index.html

```
<!DOCTYPE html>
```

Thymeleafを利用する宣言となる記述

```
<html xmlns:th="http://www.thymeleaf.org">
```

```
<head>
```

```
<title>Hello</title>
```

```
<meta charset="utf-8" />
```

```
</head>
```

<h1>内の文字列(テキストノード)を、Controllerから「message」に対応する文字列として受け取ったものに置き換える

```
<body>
```

```
<h1 th:text="${message}"></h1>
```

```
</body>
```

```
</html>
```

Thymeleafの基本文法

Controllerからの受け取り

```
<h1 th:text="${message}"></h1>
```









- **th:** → Thymeleafを使うことを示す
- **\${}** → Controllerから受け取った**変数**を表示させる

Thymeleafの基本文法

CSS、JavaScriptの表示

```
<script src="main.js" th:src="@{/main.js}"></script>
```

```
<link rel="stylesheet" href="style.css" th:href="@{/style.css}" >
```

- ▼  src/main/resources
 - ▼  templates
 -  index.html
 - ▼  static
 -  apple.jpg
 -  main.js
 -  style.css
 -  application.properties









C:\pleiades\workspace\demo\src\main\resources\static内に保存される

Thymeleafの基本文法

画像の表示

```

```

- ▼  src/main/resources
 - ▼  templates
 -  index.html
 - ▼  static
 -  apple.jpg
 -  main.js
 -  style.css
 -  application.properties

C:\pleiades\workspace\demo\src\main\resources\static内に保存される

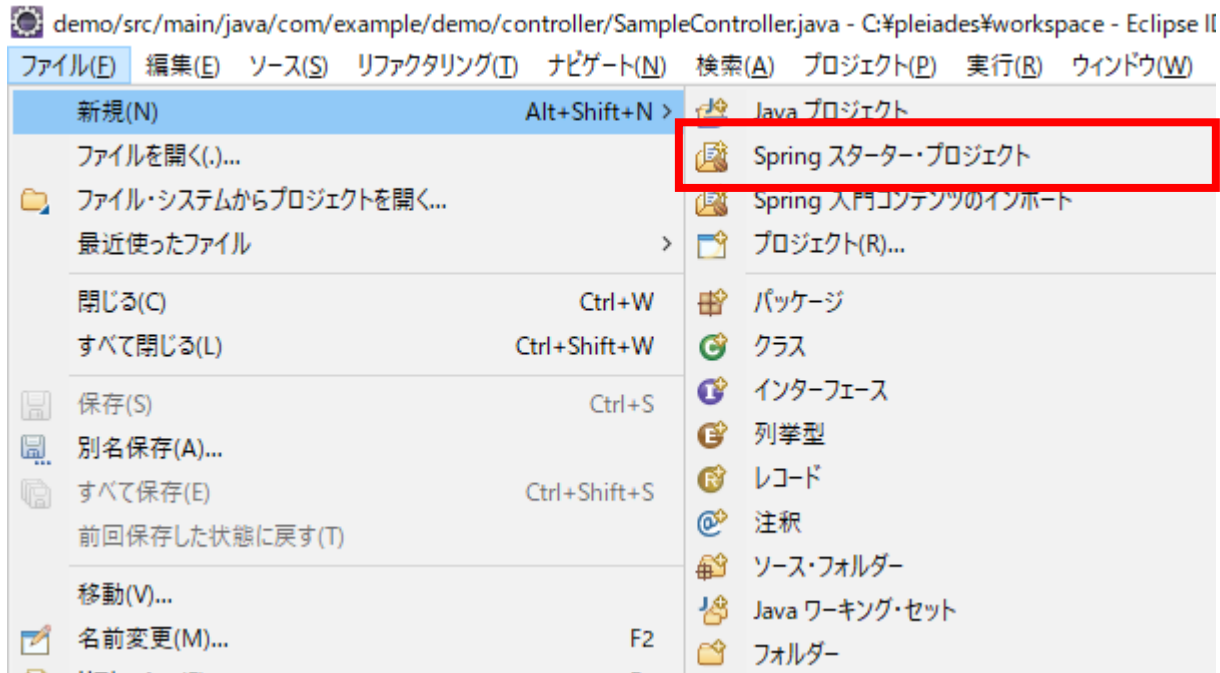
例題：画面の表示

ブラウザでlocalhost:8080/にアクセスすると、
「Hello World」と画面に表示される画面を作成しましょう



例題：画面の表示

1. アプリケーションの作成



[ファイル]>[新規]>[Springスターター・プロジェクト]を選択

例題：画面の表示

2.プロジェクトの設定

ビルドツールはGradle
Javaのバージョンは11

新規 Spring スターター・プロジェクト

サードパーティ URL:

名前:

☒ デフォルト・ロケーションを使用

ロケーション: 参照

型: パッケージング:

Java バージョン: 言語:

グループ:

成果物:

バージョン:

説明:

パッケージ:

ワーキング・セット

☐ ワーキング・セットにプロジェクトを追加(I)

ワーキング・セット(O): 新規(W)... 選択(E)...

? < 戻る(B) 次へ(N) > 完了(F) キャンセル

プロジェクト名

プログラムを配布する形式はWAR

- ・ WAR…Webアプリで利用されるクラスファイル、設定ファイル、HTMLファイル、JAR形式のライブラリなどがまとまっている
- ・ JAR…Java プログラムの実行に必要なクラスファイルや設定ファイルがまとまっている

例題：画面の表示

3.必要なソフトを指定

新規 Spring スターター・プロジェクト依存関係

Spring Boot バージョン: 2.2.7

使用頻度高:

- ☒ H2 Database
- ☒ Spring Boot DevTools
- ☒ Spring Web
- ☒ JDBC API
- ☐ Spring Data JDBC
- ☒ Thymeleaf
- ☐ MySQL Driver
- ☐ Spring Data JPA

使用可能:

検索する依存関係を入力

- Alibaba
- Amazon Web サービス
- 開発ツール
- Google Cloud Platform
- I/O
- メッセージング
- Microsoft Azure
- NoSQL
- Observability
- Ops
- Pivotal Cloud Foundry
- SQL
- セキュリティ
- Spring Cloud
- Spring Cloud Circuit Breaker

選択済み:

- X Spring Boot DevTools
- X JDBC API
- X H2 Database
- X Thymeleaf
- X Spring Web

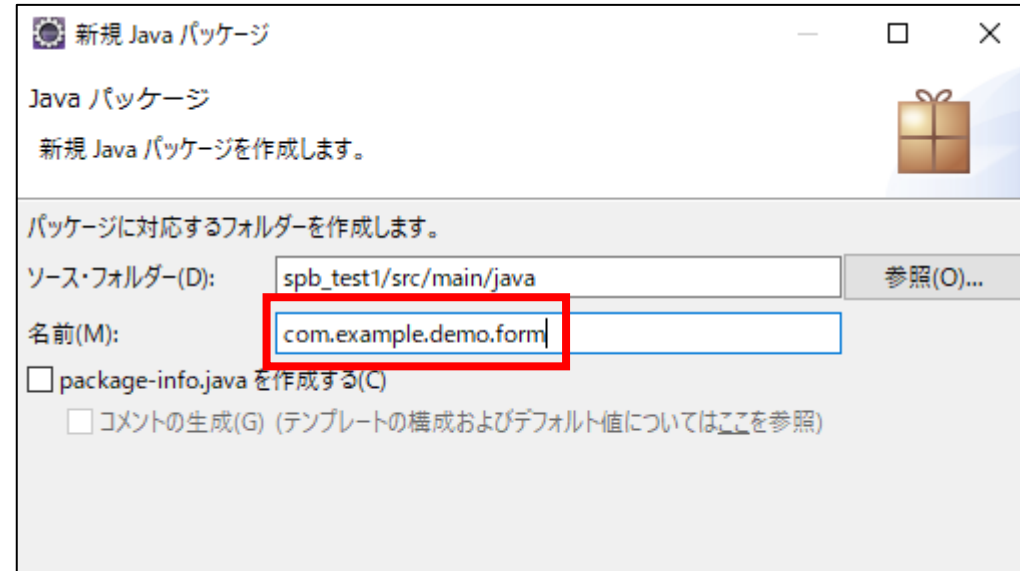
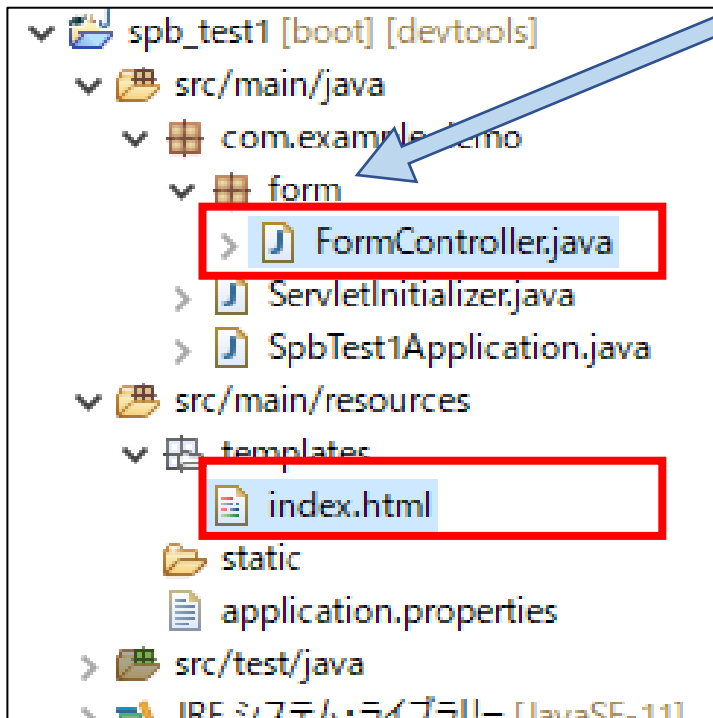
デフォルトにする 選択をクリア

? < 戻る(B) 次へ(N) > **完了(E)** キャンセル

カテゴリ	選択する項目
開発ツール	Spring Boot DevTools
I/O	検証
SQL	JDBC API
SQL	H2 Database
テンプレート・エンジン	Thymeleaf
Web	Spring Web

例題：画面の表示

4. ファイル作成、記述



↑ formパッケージを作るときの注意

FormController.java・・・Controllerの作成

index.html・・・Viewの作成

例題：画面の表示

FormController.java

```
package com.example.demo.form;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class FormController {
    @RequestMapping("/")
    public String top(Model model) {
        model.addAttribute("title", "Hello World");
        return "index";
    }
}
```

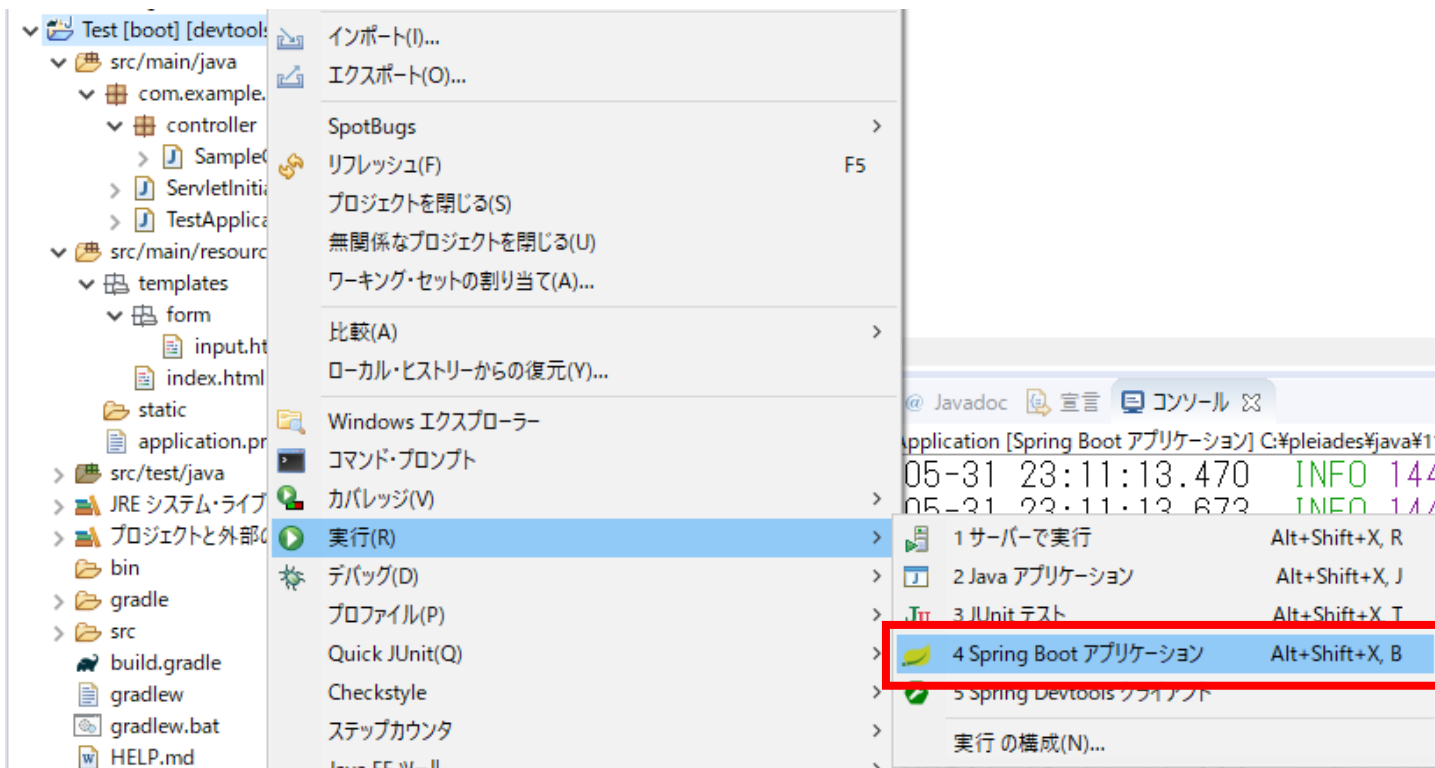
例題：画面の表示

index.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
  <head>
    <meta charset="utf-8" />
  </head>
  <body>
    <h1 th:text="${title}"></h1>
  </body>
</html>
```


例題：画面の表示

5.実行



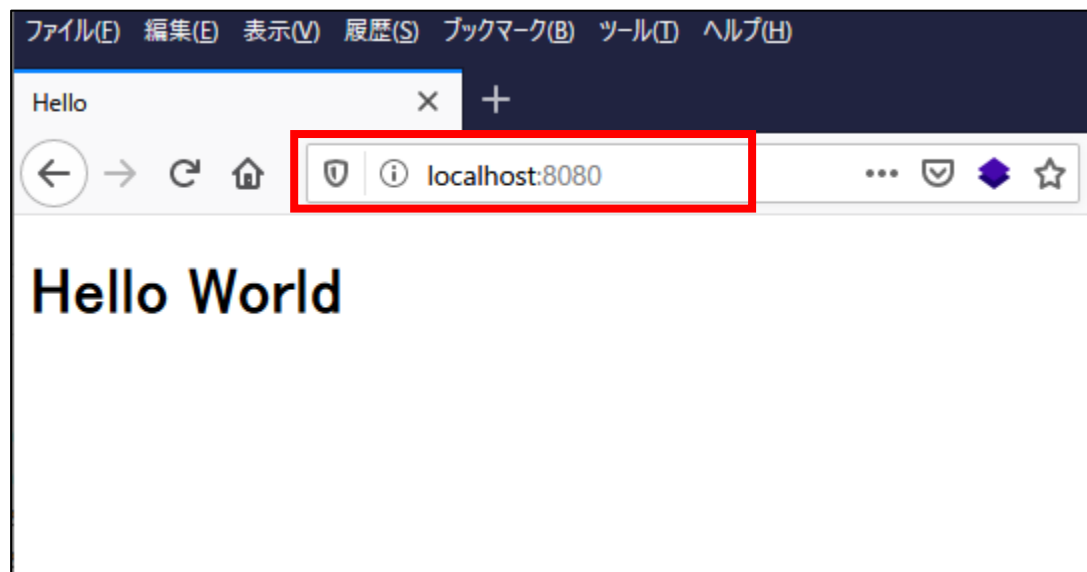
プロジェクトを右クリック→[実行]>[Spring Bootアプリケーション]をクリック

例題：画面の表示

6. ブラウザでアクセス

「localhost:8080/」でアクセスすると、表示される

- Tomcatのデフォルトのポート番号が8080
- Controllerのマッピングにおいて「@RequestMapping("/")」としているため



実践：お問い合わせフォームの入力画面の作成

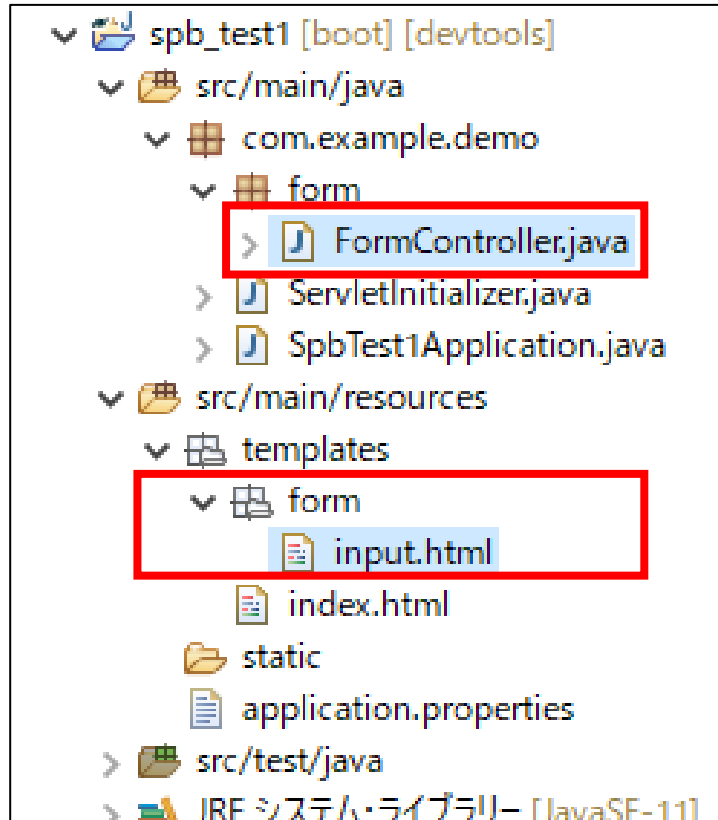
ブラウザで「**localhost:8080/form**」にアクセスすると、名前を入力するテキストボックスと送信ボタンが表示される画面を作成しましょう。



The screenshot shows a web browser window with a dark blue header bar containing menu items: ファイル(F), 編集(E), 表示(V), 履歴(S), ブックマーク(B), ツール(T), ヘルプ(H). Below the header is a tab labeled 'Hello' with a close button (X) and a new tab button (+). The address bar shows the URL 'localhost:8080/form' with navigation icons (back, forward, refresh, home) and security icons (lock, shield, star). The main content area displays the title 'サンプルフォーム' in large black text. Below the title is a label '名前:' followed by a text input field. At the bottom left is a button labeled '送信'.

実践：お問い合わせフォームの入力画面の作成

- ・フォルダ構成



FormControlController.java・・・Controllerにソースコードを追加

formフォルダ内にinput.html・・・Viewの作成

実践：お問い合わせフォームの入力画面の作成

FormController.java

```
:
@Controller
public class FormController {
    :
    :
    @RequestMapping("/form")
    public String form(Model model) {
        model.addAttribute("title","サンプルフォーム");
        return "form/input";
    }
}
```

実践：お問い合わせフォームの入力画面の作成

form/input.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
...
<body>
  <h1 th:text="${title}"></h1>
  <form method="get" action="#">
    <p>名前 : <input type="text" name="name1"></p>
    <input type="submit" value="送信">
  </form>
</body>
</html>
```

ブラウザ確認

最後にブラウザで「**localhost:8080/form**」にアクセスして、名前を入力するテキストボックスと送信ボタンが表示される画面が表示されることをチェックしましょう



A faint, light blue world map is visible in the background of the slide, centered behind the title text.

基礎編 2 (View + Controller + Model)

基礎編 2 (View + Controller + Model)

- ページ遷移
- フォームによるデータの送付
- 実践：お問い合わせフォームの確認画面の作成

ページ遷移(リンク)

ページA

[次のページへ](#)

form.html



<a>による画面遷移
を行えばよい

ページB

ページBです

confirm.html

<a>によるリンク

index.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="utf-8" />
</head>
<body>
  <h1>トップページ</h1>
  <a href="confirm.html">確認画面へのリンク</a>
</body>
</html>
```

ビルドによるコードの変化

ビルド前のindex.html

```
<a href="abc.html" th:href="@{sample2}">sample2へのリンク</a>
```

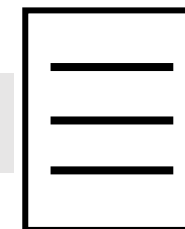


ビルドして実行

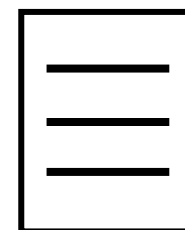
ビルド後のindex.html

```
<a href="sample2">sample2へのリンク</a>
```

SpringBootアプリケーションとして実行すると、
「th:●●」が解釈されて、属性が上書きされる



そのまま解釈される
(JSFではこれができなかった)



SpringBootアプリケーションとして実行すると、
Thymeleafの部分が書き変わる

ページ遷移(フォーム)

入力画面

名前

form.html



ユーザーの入力データ
がある場合は、
<a>の画面遷移でなく
<form>で送る必要が
ある

確認画面

名前 田中

confirm.html

ページ遷移(フォーム)

階層構造



FormController.java

```
:  
  
@Controller  
public class FormController {  
    :  
    :  
    @RequestMapping("/form")  
    public String form(Model model) {  
        model.addAttribute("title","サンプルフォーム");  
        return "form/input";  
    }  
}
```

form/input.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
...
<body>
  <h1 th:text="${title}"></h1>
  <form method="get" action="#" th:action="@{/confirm}">
    <p>名前 : <input type="text" name="name1"></p>
    <input type="submit" value="送信">
  </form>
</body>
</html>
```


Form.java

```
package com.example.demo.form;

public class Form {
    private String name1;    . . . HTMLファイルのname属性の値と一致させる
    public Form() { }    . . . コンストラクター（何もしない）
    public String getName1() {
        return name1;
    }
    public void setName1(String name1) {
        this.name1 = name1;
    }
}
```

ゲッター

セッター

FormController.javaの変更 ※この状態ではエラーが出ます

```
@Controller
public class FormController {
    :
    :
    @RequestMapping("/confirm")
    public String confirm(Model model) {
        model.addAttribute("title","確認ページ");
        return "form/confirm";
    }
}
```

追記

confirm.htmlを作成（データの受け取りまで）

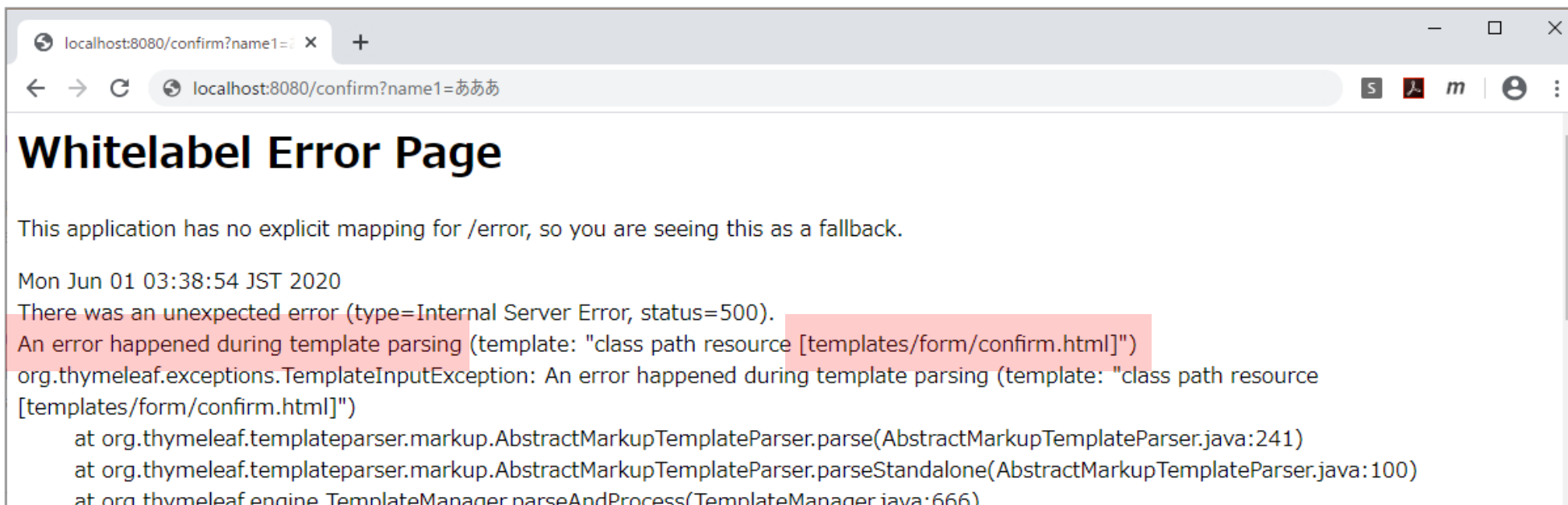
```
<h1><span th:text="${title}"></span></h1>
<div>
  <p>入力内容ここから</p>
  <p th:text="${form.name1}"></p>
  <p>入力内容ここまで</p>
</div>
```

ブラウザでプレビューしてみましょう。

`${form.name1}`によってForm.javaのgetName1を呼び出そうとしますが、エラーが出ます。

ページ遷移

しかし、confirm.htmlのためにエラーが表示されてしまいます



⋮

「`th:text="${form.name1}"`」というThymeleafに対応するデータがControllerから送られていないため、エラーが表示されます

ページ遷移

FormController.javaの変更2 ※エラーが解消されます

```
@Controller
public class FormController {
    ...
    @RequestMapping("/confirm")
    public String confirm(Model model , Form form) {
        model.addAttribute("title","確認ページ");
        return "form/confirm";
    }
}
```

追記

confirm.htmlを表示する際に、
Formクラスのオブジェクトを渡せる
ようになります
※注意※
クラス名を小文字にしたオブジェクト名
にしなければなりません。

ブラウザでプレビューしてみましょう。

Formクラスのオブジェクトが「form」として渡せているはずです。

ページ遷移

confirm.htmlの変更

```
<h1><span th:text="${title}"></span></h1>
```

```
<div th:object="${form}"> ————— formを受け取る
```

```
<p>入力内容ここから</p>
```

```
<p th:text="*{name1}"></p>
```

```
<p>入力内容ここまで</p>
```

```
</div>
```

(Formクラスのオブジェクト
を受け取る)

*{}で、親要素で受け取っているオブジェクト
のメンバーにアクセスできる。

(\${form.name1}と書いてもOK)

ページ遷移

confirm.htmlを変更

```
<h1><span th:text="\${title}"></span></h1>
```

```
<div th:object="\${form}">
```

```
  <p>入力内容ここから</p>
```

```
  <p th:text="\*{name1}"></p>
```

```
  <p>入力内容ここまで</p>
```

```
  <form method="get" action="#" th:action="@{/form}">
```

```
    <input type="hidden" name="name1" th:value="\*{name1}">
```

渡したいデータ

```
    <input type="submit" value="入力画面へ">
```

```
  </form>
```

```
  <form method="get" action="#" th:action="@{/complete}">
```

```
    <input type="hidden" name="name1" th:value="\*{name1}">
```

渡したいデータ

```
    <input type="submit" value="完了画面へ">
```

```
  </form>
```

```
</div>
```

入力画面

戻る

完了画面

進む

このままではエラーが起きます。他のファイルを変更して解決をします

ページ遷移（エラー解決のヒント）

form.htmlを**変更**

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
...
<body>
  <h1><span th:text="${title}"></span></h1>
  <form method="get" action="#" th:action="@{form/confirm}" th:object="${form}">
    <p>名前 : <input type="text" name="name" th:value="*{name1}"></p>
    <input type="submit" value="送信">
  </form>
</body>
</html>
```

formを受け取る
(form に格納されたForm
オブジェクトを受け取る)

追記

***{}で、親要素で受け取っているオブジェクト
のメンバーにアクセスできる。**
(ここでは\${form.name1}と書いても同じ)

バリデーション（入力値の検証）

バリデーションとは、入力チェックのこと。

たとえば、ECサイトで10個以上の購入は不可とするなど、ユーザーの入力を制限したいときに使います。

※注意点※

SpringBootのバージョン2.3以降から、
「検証」を選択しておく必要があります
これがないと、javax.validation*の
ライブラリが利用できません

新規 Spring スターター・プロジェクト依存関係

Spring Boot バージョン: 2.4.3

使用頻度高:

☒ H2 Database ☒ JDBC API ☒ Spring Boot DevTools
☒ Spring Web ☒ Thymeleaf ☒ 検証

使用可能: 検索する依存関係を入力

Alibaba
Amazon Web サービス
開発ツール
Google Cloud Platform
I/O
メッセージング
Microsoft Azure
NoSQL
Observability
Ops
SQL

選択済み:

X Spring Boot DevTools
X 検証
X JDBC API
X H2 Database
X Thymeleaf
X Spring Web

デフォルトにする 選択をクリア

? < 戻る(B) 次へ(N) > 完了(F) キャンセル

バリデーション（入力値の検証）

設定方法

1. Form.javaにバリデーション専用のアノテーションを付ける
2. FormController.javaにエラー時に入力画面に戻す処理を追加
3. input.htmlに、エラー内容を表示させる記述を追加

バリデーション（入力値の検証）

Form.java

```
public class Form {  
    @Size(min=1, max=10, message="1~10文字以内になしてください")  
    private String name;  
  
    public Form() {}  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

name1からnameに修正しています
(input.html、confirm.htmlも修正が必要です)

バリデーションのアノテーション

アノテーション	説明
@AssertFalse	falseであることをチェック
@AssertTrue	trueであることをチェック
@DecimalMax	Decimal型が最大値以下であることをチェック
@DecimalMin	Decimal型が最小値以上であることをチェック
@Digits(integer= ,fraction=)	数値の桁数が範囲内であることをチェック ※integerは整数、fractionは小数の最大桁数
@Past	過去の日付であることをチェック
@Future	未来の日付であることをチェック
@Max	数値が最大値以下であることをチェック
@Min	数値が最小値以上であることをチェック
@NotNull	Nullでないことをチェック
@Null	Nullであることをチェック

バリデーションのアノテーション

アノテーション	説明
@Pattern(regex= , flag=)	指定した正規表現を満たすことをチェック
@Size(min= , max=)	文字数、またはCollectionなどの数が範囲内であることをチェック
@Valid	ネストしたクラスのバリデーションを実行する。
@CreditCardNumber	クレジットカード番号が妥当かチェック
@Email	メールアドレスが妥当（RFC2822）かチェック
@Length(min=, max=)	文字数の範囲をチェック
@NotBlank	空文字またはNullでないことをチェック
@NotEmpty	文字またはCollectionなどがNullまたは空でないことをチェック
@Range(min= ,max=)	数値の範囲をチェック


バリデーション（入力値の検証）

FormController.java

```
@Controller
public class FormController {
    :
    @RequestMapping("/form")
    public String form(Form form) {
        return "form/input";
    }
    @RequestMapping("/confirm")
    public String confirm(@Validated Form form, BindingResult result, Model model) {

        if(result.hasErrors()) {
            model.addAttribute("title","入力ページ");
            return "form/input";
        }

        model.addAttribute("title","確認ページ");
        return "form/confirm";
    }
}
```



もしエラーがあれば、
入力画面のページにジャンプさせる

バリデーション（入力値の検証）

FormController.java

```
@Controller
public class FormController {
    :
    @RequestMapping("/form")
    public String form(Form form) {
        return "form/input"; バリデーション済みのデータ エラー情報が保存されている
    }
    @RequestMapping("/confirm") /
    public String confirm(@Validated Form form, BindingResult result, Model model) {

        if(result.hasErrors()) {
            model.addAttribute("title","入力ページ");
            return "form/input";
        }

        model.addAttribute("title","確認ページ");
        return "form/confirm";
    }
}
```

もしエラーがあれば、
入力画面のページにジャンプさせる

バリデーション（入力値の検証）

FormController.java

```
@Controller
public class FormController {
    :
    @RequestMapping("/form") /
    public String form(Form form) {
        return "form/input"; バリデーション済みのデータ エラー情報が保存されている
    }
    @RequestMapping("/confirm") /
    public String confirm(@Validated Form form, BindingResult result, Model model) {

        if(result.hasErrors()) {
            model.addAttribute("title","入力ページ");
            return "form/input";
        }

        model.addAttribute("title","確認ページ");
        return "form/confirm";
    }
}
```

エラー時にエラーメッセージを表示
させるために入力データを渡す

もしエラーがあれば、
入力画面のページにジャンプさせる

バリデーション（入力値の検証）

form.html

```
<form method="get" action="#" th:action="@{/confirm}" th:object="${form}">
  <p>名前 : <input type="text" name="name" th:value="*{name}"> </p>
  <div th:if="${#fields.hasErrors('name')}" th:errors="*{name}">エラー</div>
  <input type="submit" value="送信">
</form>
```

Formクラスのオブジェクトのフィールド変数
nameでエラーが起きているかのチェック

エラー内容の出力

バリデーション（入力値の検証）

form.html

```
<form method="get" action="#" th:action="@{/confirm}" th:object="${form}">
  <p>名前 : <input type="text" name="name" th:value="*{name}"> </p>
  <div th:if="${#fields.hasErrors('name')}" th:errors="*{name}">エラー</div>
  <input type="submit" value="送信">
</form>
```

入力した内容を表示

入力データやエラーメッセージが入っている

Formクラスのオブジェクトのフィールド変数

nameでエラーが起きているかのチェック

エラー内容の出力

【補足】自作のバリデーションを設置

その他の方法として、**ビーンバリデーション**を利用することも可能。

「ValidationMessages.properties」を作って、各アノテーションのデフォルトメッセージを変更する。src/main/resourcesの直下、application.propertiesと同じところにValidationMessages.propertiesを作成すれば使える。

ValidationMessages.properties

```
javax.validation.constraints.NotBlank.message=必須入力です
```

```
javax.validation.constraints.Size.message=文字数が範囲外です
```

バリデーションの演習

- ・ 数字を入力し、10以上でエラーが出るようにする
- ・ 郵便番号の入力項目を作って、正規表現でチェックする
- ・ メールアドレスの入力項目を作り、チェックをする(@Emailで)
- ・ メールアドレスの入力項目を作り、正規表現を使ってチェックする

バリデーションの解答

Form.java

```
public class Form {  
    @Size(min=1,max=10,message="1～10文字以内に入してください")  
    private String name;  
  
    @Max(value=9,message="10文字未満で入力してください")  
    private int num;  
  
    @Email  
    private String email1;  
  
    @Pattern(regex="^[a-z0-9]{1,}¥¥@[a-z0-9]{1,}¥¥¥¥.[a-z]{1,}$"  
        , message="正しいメールアドレスの形式で入力してください")  
    private String email2;  
}
```

¥2つで1つの扱いとなる

/

|

バリデーションの解答

form/input.html

```
<p>名前 : <input type="text" name="name" th:value="*{name}"></p>  
<div th:if='${#fields.hasErrors("name")}' th:errors="*{name}">エラー</div>
```

```
<p>数字 : <input type="text" name="num" th:value="*{num}"></p>  
<div th:if='${#fields.hasErrors("num")}' th:errors="*{num}">エラー</div>
```

```
<p>郵便番号 : <input type="text" name="zipcode" th:value="*{zipcode}"></p>  
<div th:if='${#fields.hasErrors("zipcode")}' th:errors="*{zipcode}">エラー</div>
```

```
<p>メール1 : <input type="text" name="email1" th:value="*{email1}"></p>  
<div th:if='${#fields.hasErrors("email1")}' th:errors="*{email1}">エラー</div>
```

```
<p>メール2 : <input type="text" name="email2" th:value="*{email2}"></p>  
<div th:if='${#fields.hasErrors("email2")}' th:errors="*{email2}">エラー</div>
```