

6.3 Spring Boot 基礎

- ・プロジェクト作成
- ・初めてのサーバー作成
- ・Thymeleaf



目 次

1

プロジェクト作成

2

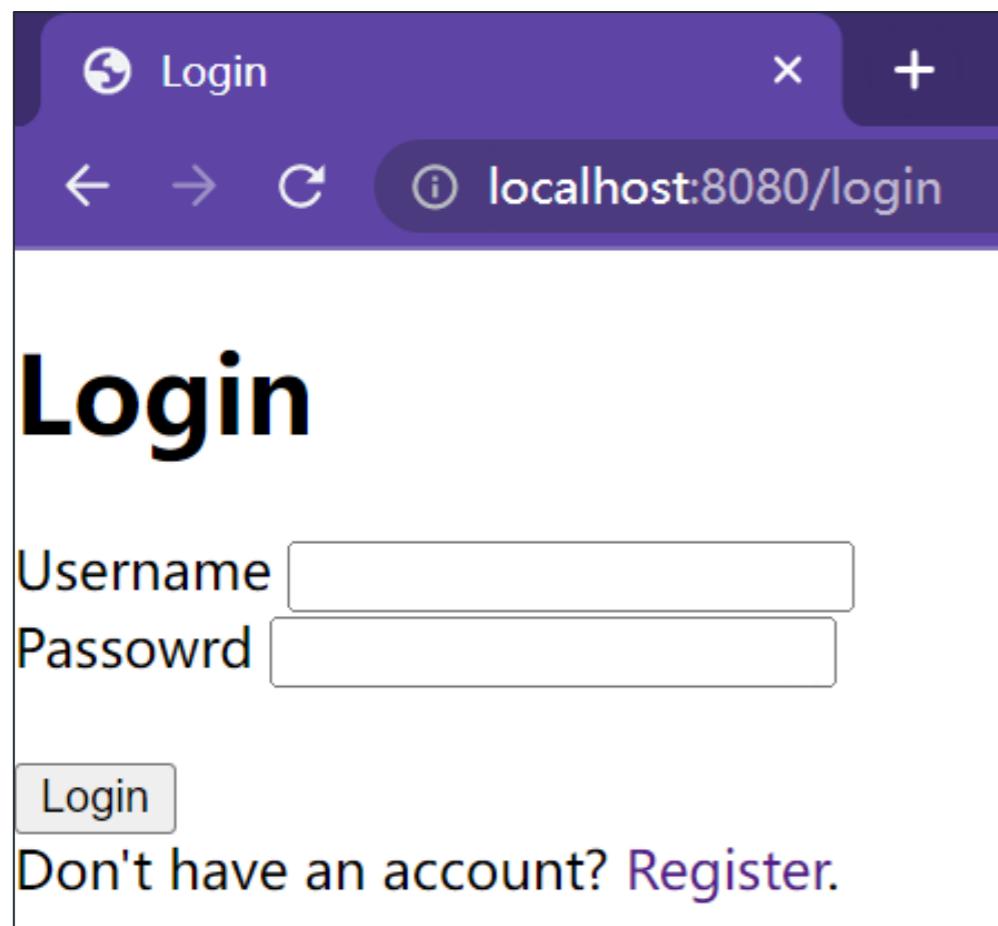
初めてのサーバ作成

3

Thymeleaf

目標: 簡単なログインサイト

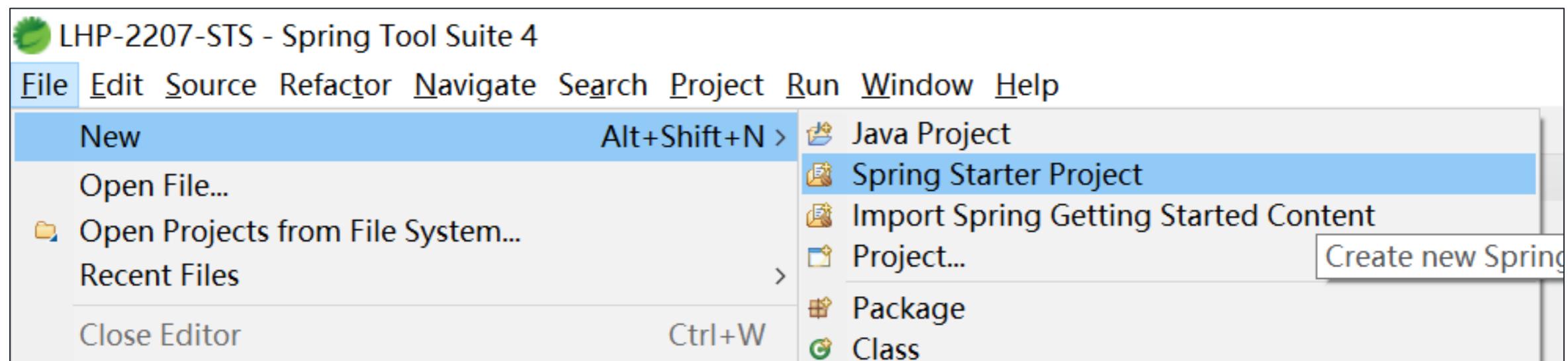
- 今回の授業の目標は、データベースとのやりとりがない、簡単なログインウェブサイトを作成することです。



- まず、Spring Boot の新規プロジェクトを作りましょう。

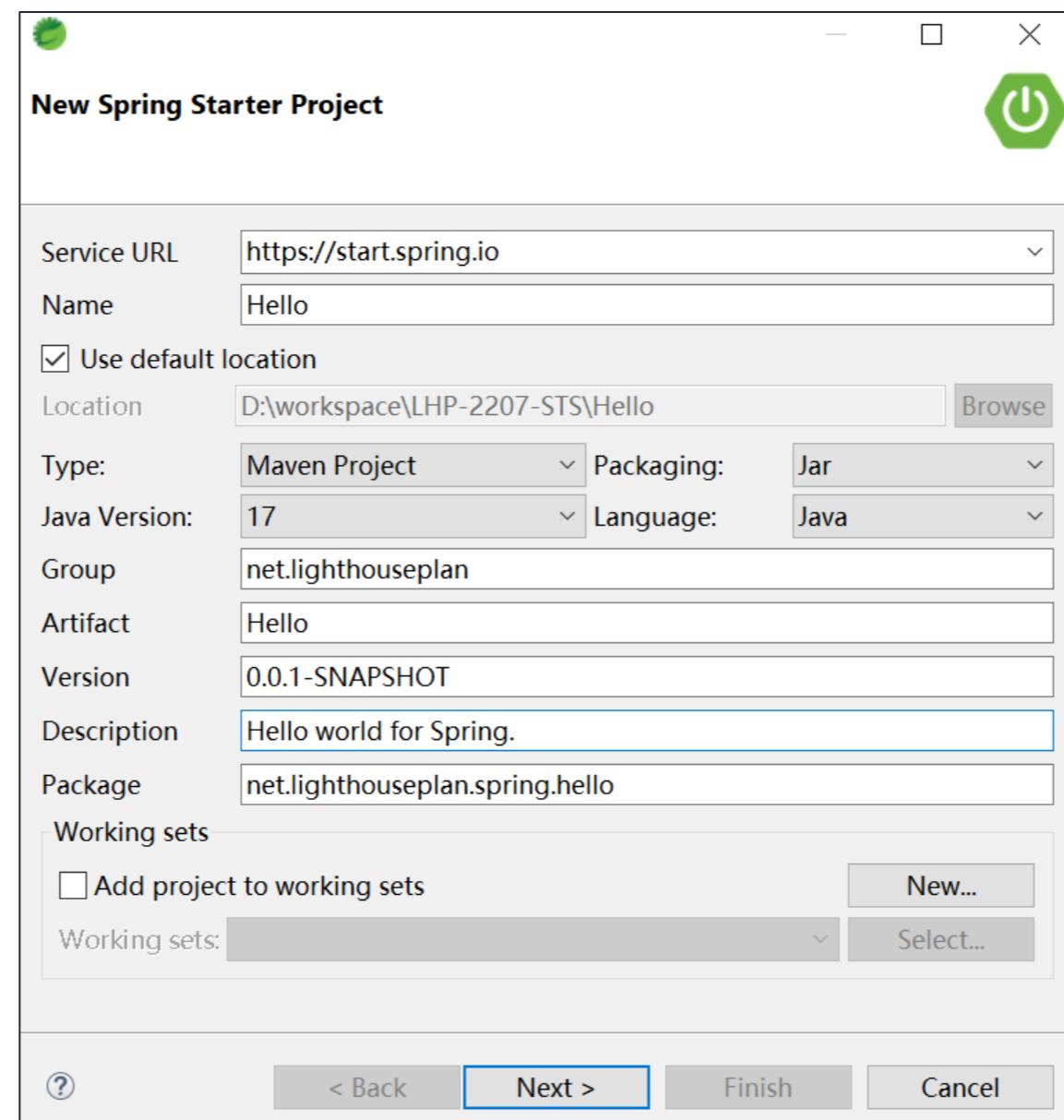
ステップ1: プロジェクトの作成

- ディレクトリエリアにある Create new Spring Starter Project をクリック。
- このオプションがない場合（既存のプロジェクトがある場合など）、メニューバーの File → Spring Starter Project をクリックします。



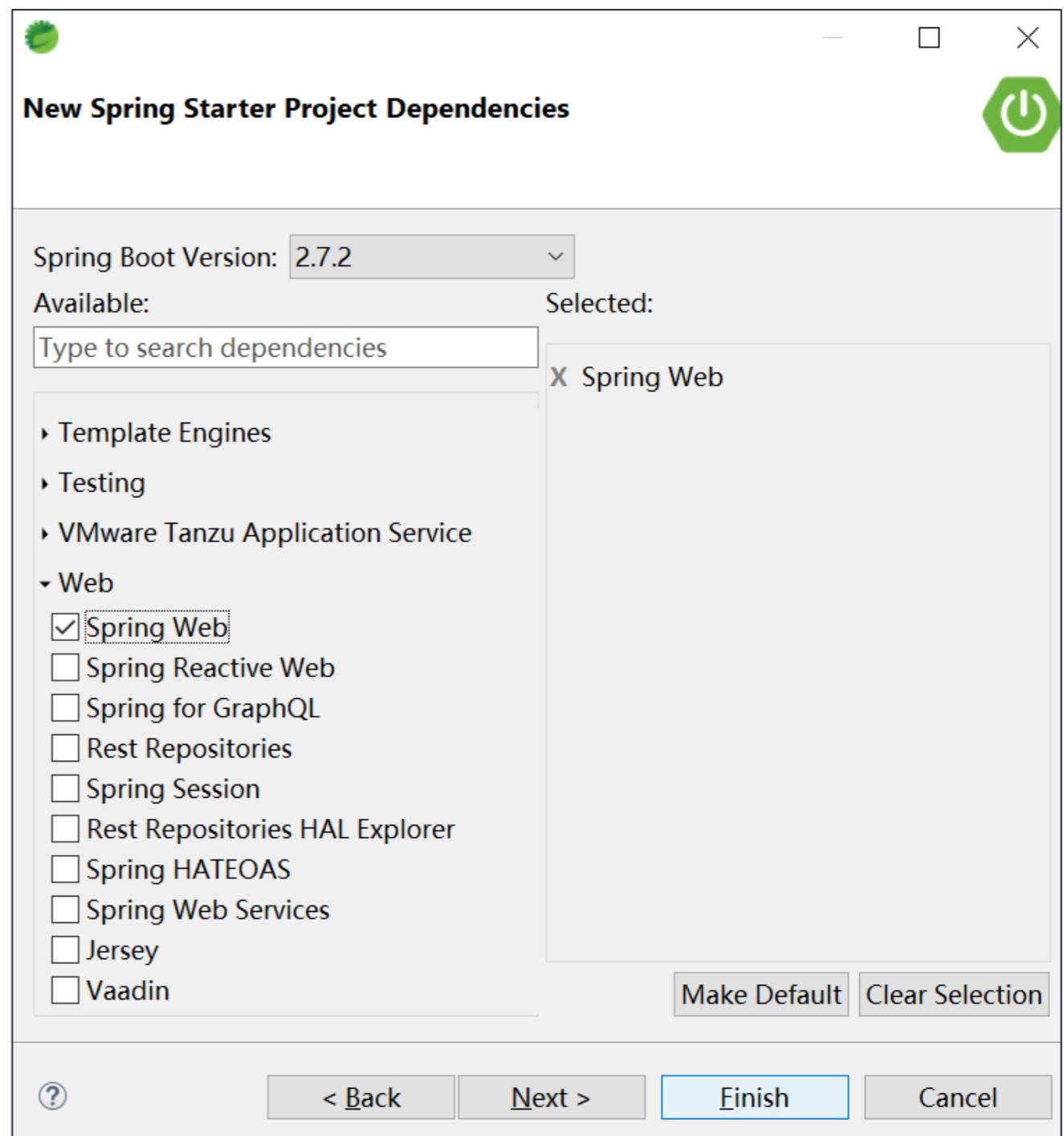
ステップ 2: プロジェクト情報の設定

- 必要に応じてプロジェクトの情報を設定して、Next。

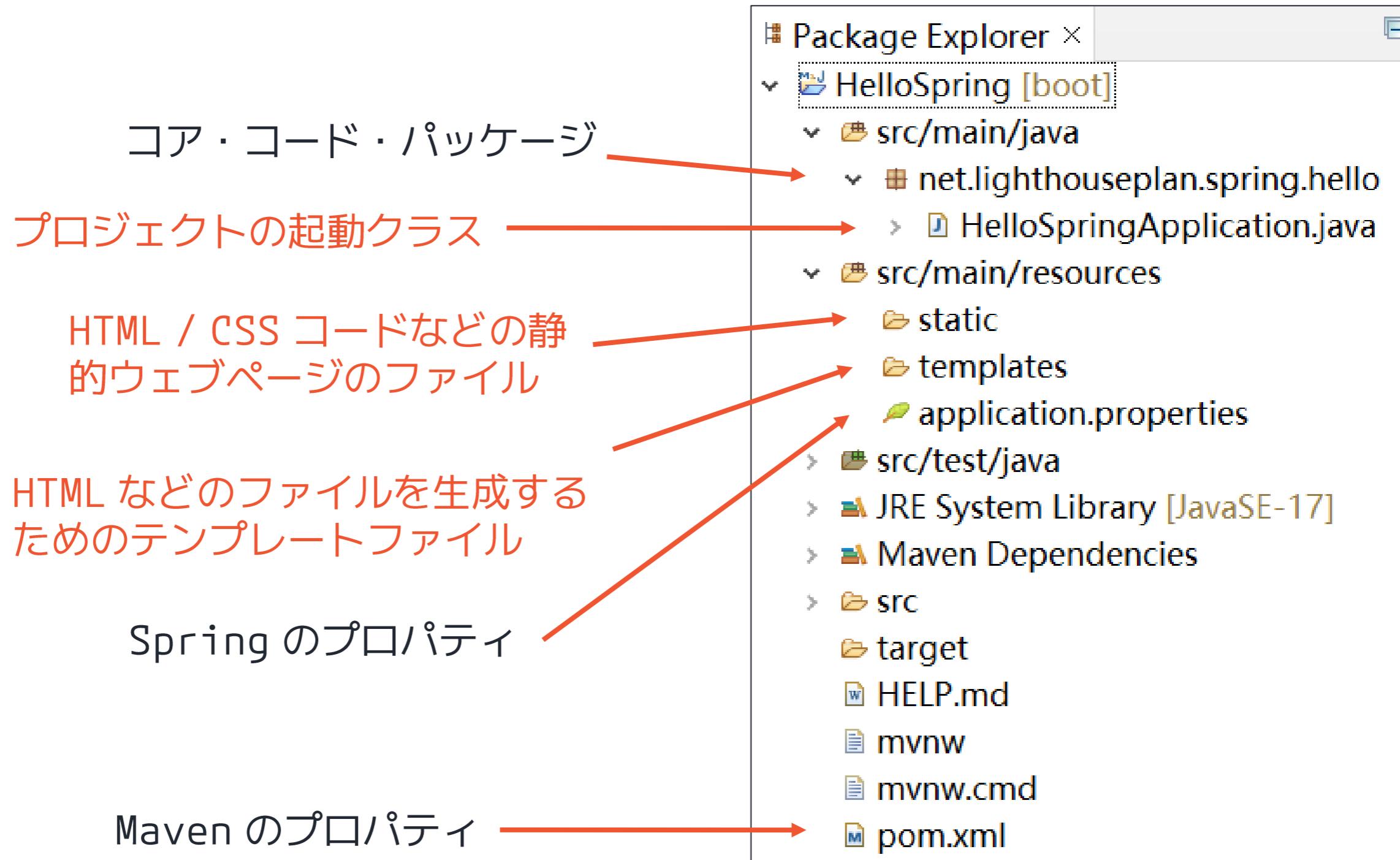


ステップ 3: 依存関係の追加

- ここでは、必要な依存関係を検索し、プロジェクトが作る時点で自動的に追加させることができます。
- もちろん、Maven を使って後から依存関係を追加しても構いません。
- ここで、Web → Spring Web の依存関係を追加し、Finish をクリック。



プロジェクト構成



参考資料

- Spring Boot 公式 API :
 <https://docs.spring.io/spring-boot/docs/current/api/>
- Spring Boot 日本語版リファレンスマニュアル:
 <https://spring.pleiades.io/projects/spring-boot>
- Spring Boot 中国語版リファレンスマニュアル:
 <https://www.springcloud.cc/spring-boot.html>



Q & A

Question and answer



1 プロジェクト作成

2 初めてのサーバ作成

3 Thymeleaf

プロジェクト開始のプロセス

- STS で作成した Spring Starter プロジェクトには、最初から Java のコードファイルが付属しています。
- 「[プロジェクトの名前]Application.java」を開いてください。ご覧のように、このファイルには Spring Boot プロジェクトを起動する main メソッドがすでに用意されています。今のところ、**変更する必要はありません**：

```
1 @SpringBootApplication
2 public class HelloSpringApplication {
3
4     public static void main(String[] args) {
5         SpringApplication.run(HelloSpringApplication.class, args);
6     }
7
8 }
```

最も基本的なコントローラー

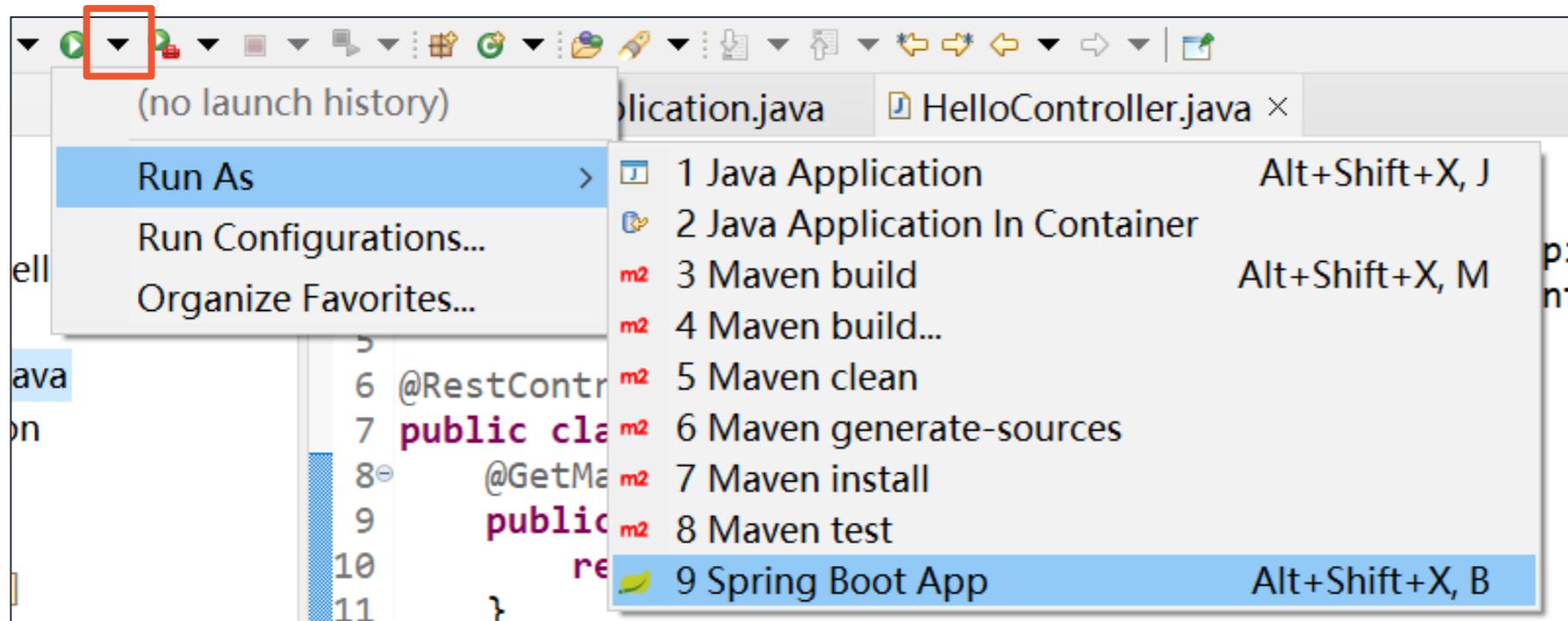
- `HelloController.java` という Java コードファイルを作成し、以下の内容を追加します：

```
1 @RestController
2 public class HelloController {
3     @GetMapping("/")
4     public String index() {
5         return "Hello, world!";
6     }
7 }
```

- クラスとメソッドの前のアノテーション (`@RestController` と `@GetMapping`) に注意。
- アノテーションでコンパイルエラーがある場合、STS に該当するパッケージをインポートさせてください。

プロジェクトの起動

- プロジェクトを右クリック、Run ボタンの右の三角をクリックし、Run As → Spring Boot App:

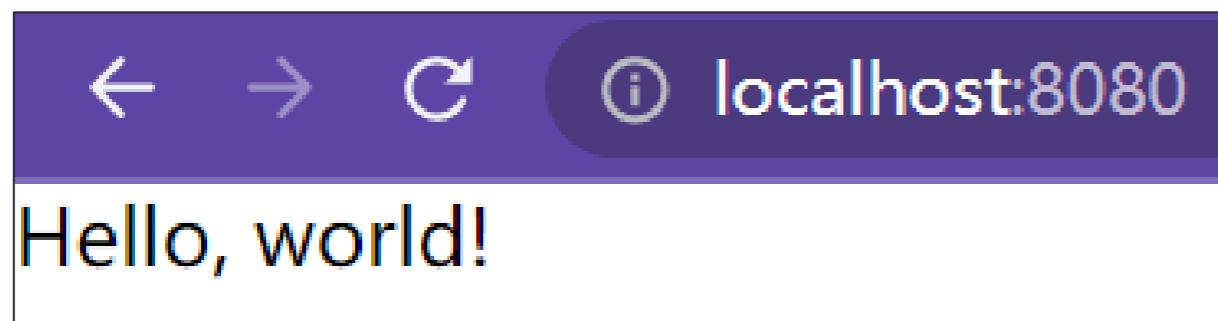


- サーバーの起動が終わるまで待ちます。起動できたら、コンソールに以下のような出力が表示されます:

```
...  
: Started HelloSpringApplication in 1.525 seconds (JVM running for 2.092)
```

結果のチェック

- ブラウザで次のリンクを開いてください:
<http://localhost:8080/>
- 以下のような結果が表示されるはずです:



HelloController の解説: @RestController

- `@RestController` アノテーションは、Spring にこのクラスがコントローラであることを知らせ、HTTP リクエストを処理するためのいくつかのメソッドを含むべきです。
- ブラウザがある URL にアクセスすると、該当する URL に対して HTTP リクエストを送信します。コントローラはそれを受け取って、ブラウザに表示したいもの（HTML コード）を文字列として返します。

次へ 

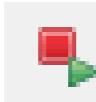
 前へ

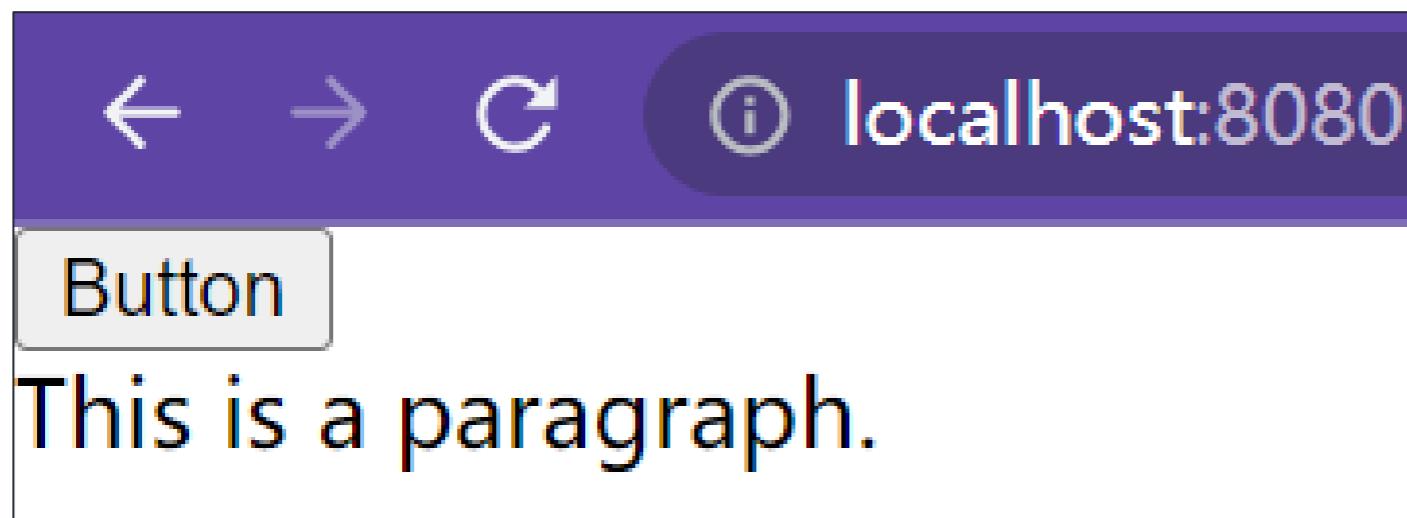
- `index()` メソッドは、コントローラから返される内容を定義します。戻り値は、任意の HTML コード を記述することができます。例えば：

```
1 @RestController
2 public class HelloController {
3     @GetMapping("/")
4     public String index() {
5         return "<button>Button</button><p>This is a paragraph.</p>";
6     }
7 }
```

- 変更後のサーバをテストするには、プロジェクトを**再起動する必要があります。**

プロジェクトの再起動

- 変更したプロジェクトを起動する前に、必ず  をクリックし、先に起動した項目を閉じます。
- また、  をクリックするだけでワンクリックで再起動することができます。ただし、新しいプロジェクトに切り替えるときは使えません。



HelloController の解説: @GetMapping

- `@GetMapping("/")` アノテーションは Spring に、ユーザーが URL 「`http://localhost:8080/`」にアクセスするときの HTTP リクエストに応答するなら、この `index()` メソッドを使用するように指示します。
- 「Get」は、GET メソッドによるリクエストに応答することを意味します。
- 括弧内の「`"/"`」の部分は、相対的な URL を指示します。「`localhost:8080`」以降の部分を表しています。

次へ 

 前へ

- 例えば、"/" を "/hello" に変更した場合、先ほどのページを見るには「`http://localhost:8080/hello`」にアクセスしなければなりません。このように、異なる URL へのリクエストに対応するメソッドを複数記述することが可能です：

```
1 @GetMapping("/hello")
2 public String hello() {
3     return "Hello, world!";
4 }
5
6 @GetMapping("/bye")
7 public String bye() {
8     return "Bye, world!";
9 }
```

Tips

メソッド名は URL と無関係であり、**任意**に命名できます。

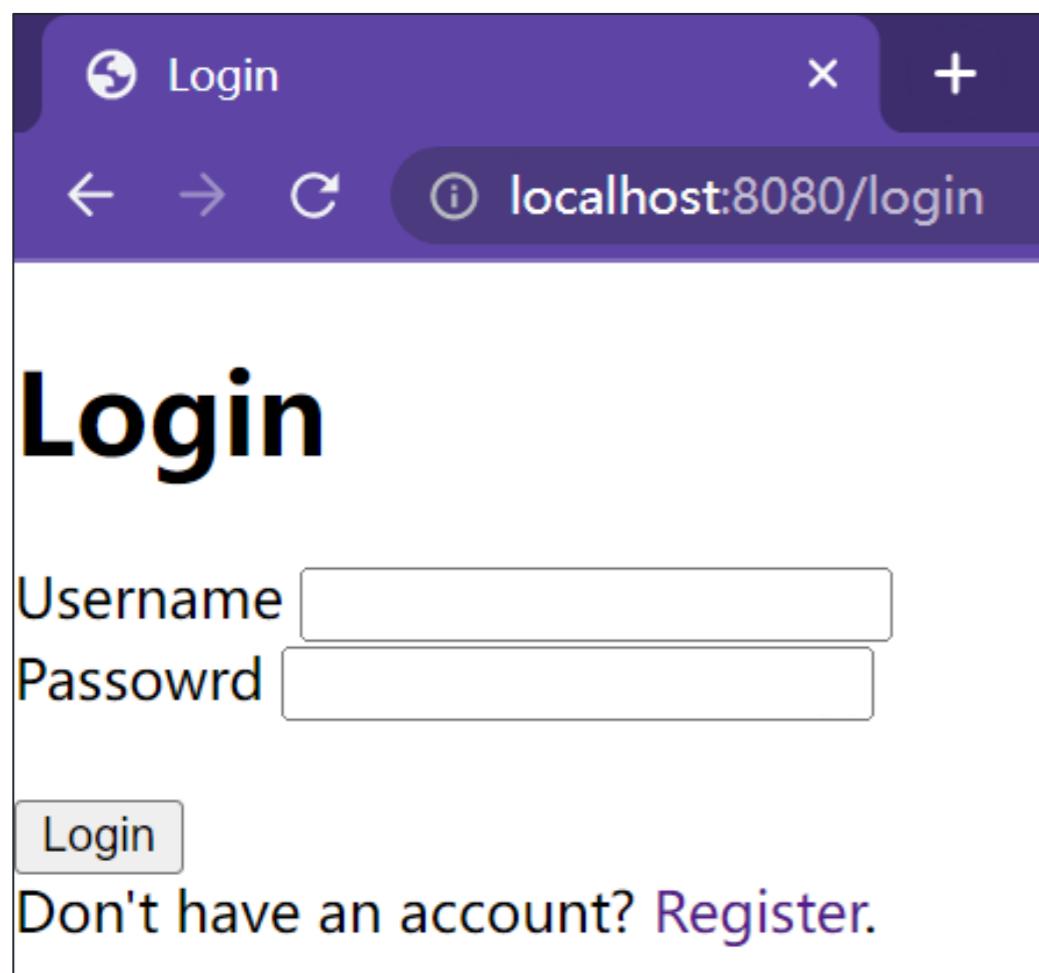
復習: HTTP リクエストのメソッド

- GET メソッドは、デフォルトで最もシンプルな HTTP リクエストのメソッドです。ブラウザのアドレスバーに直接 URL を入力してウェブページにアクセスすると、ブラウザは GET メソッドを使ってサーバからレスポンスをもらいます。
- POST メソッドは、個人情報や大量のデータをサーバに送信するため使用することができます。
- Spring では、`@GetMapping()` と `@PostMapping()` のアノテーションで、それぞれ GET と POST メソッドに対するリクエストに対応します。

次へ 

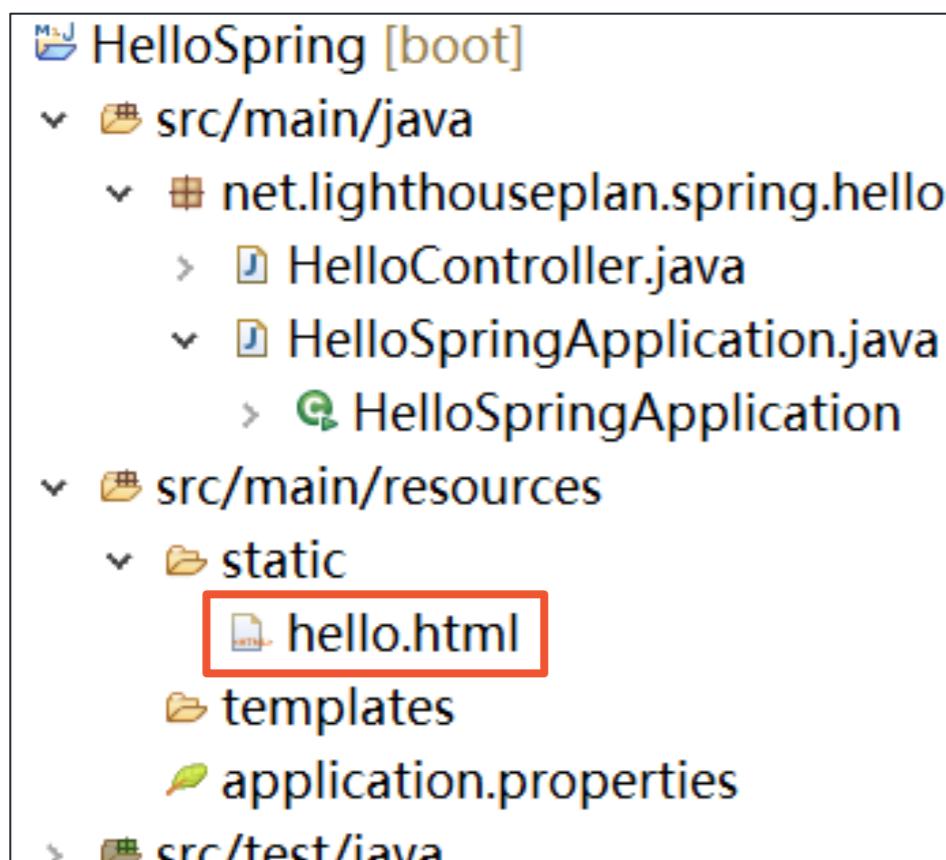
◀ 前へ

- 考えてください：ログインページでログインボタンをクリックした際に送信されるリクエストは、`@GetMapping()` と `@PostMapping()` のどちらで対応すべきなのでしょうか？



HTML ページの作成

- 今まででは文字列だけをブラウザに返せました。この方法でページ全体を返そうとすると、非常に面倒になります。書き上げた HTML ファイルを直接返せるように望みます。
- まず、static フォルダに返されたい HTML ページを作成：



hello.html:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title>Hello Spring</title>
6 </head>
7
8 <body>
9   <h1>Hello, world!</h1>
10 </body>
11 </html>
```

コントローラでページを返す

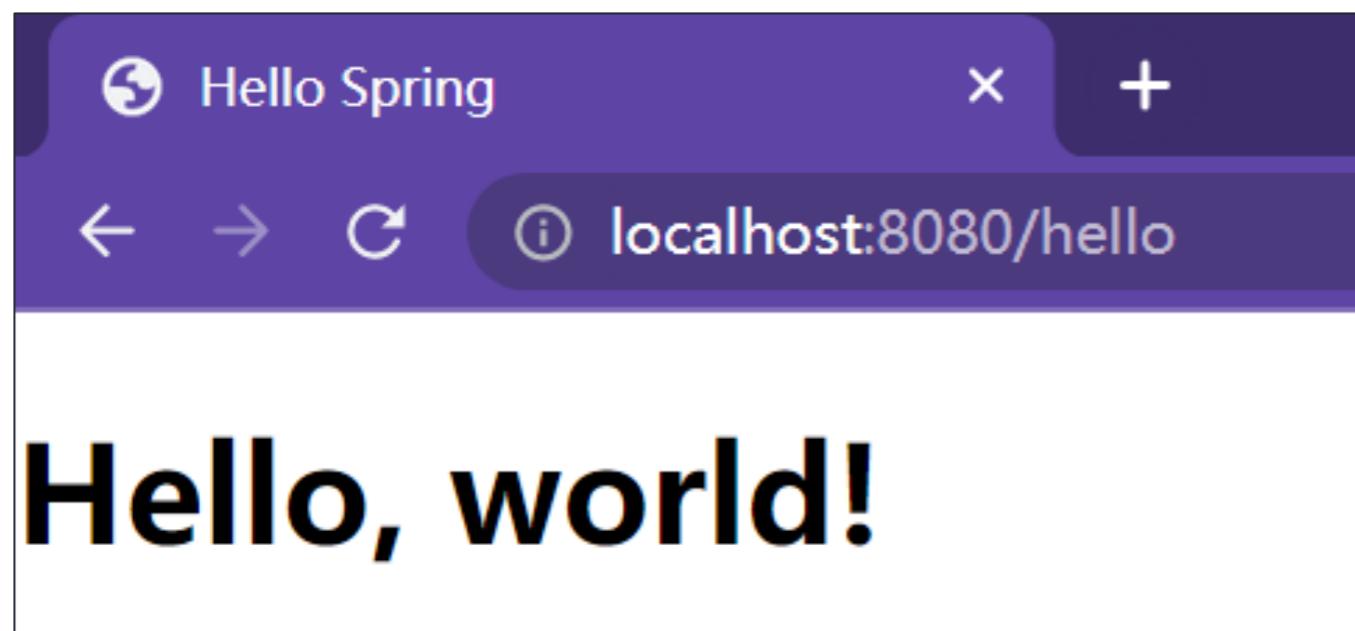
- 次に、コントローラのコードを修正する必要があります：

```
1 @Controller
2 public class HelloController {
3     @GetMapping("/hello")
4     public String hello() {
5         return "hello.html";
6     }
7 }
```

- ここで：
 - **@Controller** アノテーションは、これが HTML ページを返すコントローラであることを示すものです。
 - `hello()` メソッドの戻り値は、返したい HTML ファイルのパス。

結果のチェック

- プロジェクトを再起動し、<http://localhost:8080/hello> にアクセスし、対応するページを閲覧できます：

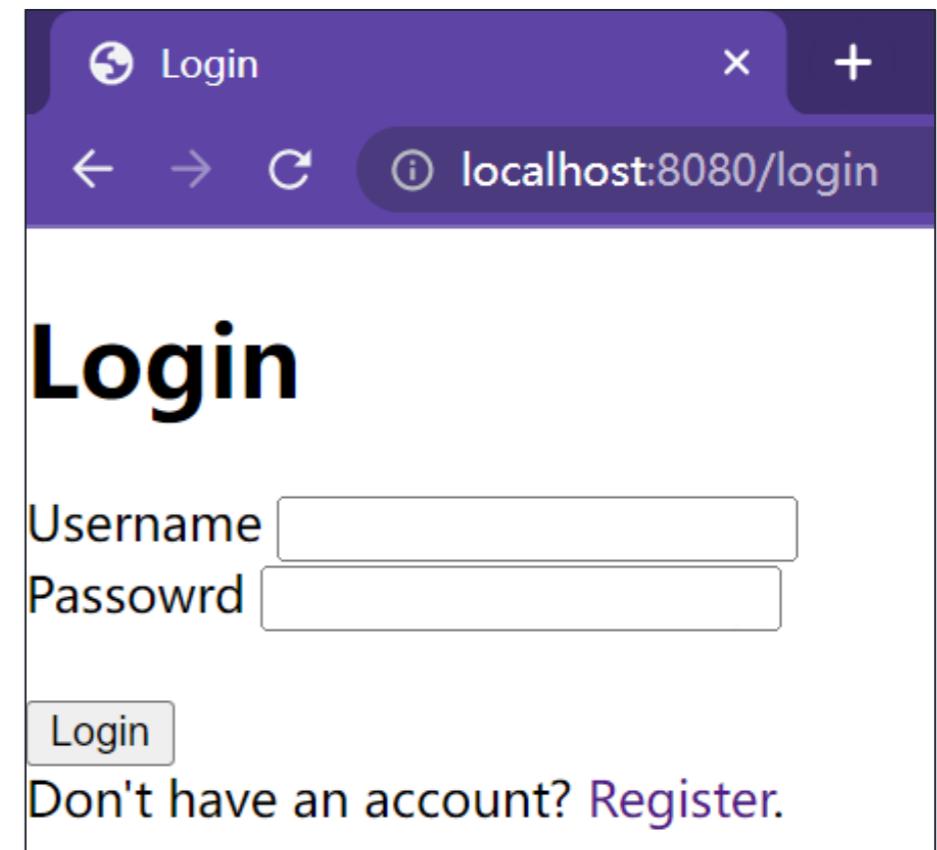




Question and answer

演習：ログイン画面の作成

- 目標：ログインページの作成。
- 右のようなログインページを作成し、`login.html` ファイルとして保存します。
- `localhost:8080/login` にアクセスした際にページが表示されるように、コントローラーのメソッドを記述してください。
- まずは、自分で試してみましょう！



The screenshot shows a web browser window with a purple header bar. The header bar contains a 'Login' icon, a close button ('x'), and a plus sign ('+') for opening new tabs. Below the header, the URL 'localhost:8080/login' is displayed. The main content area has a large 'Login' title at the top. Below it are two input fields: one for 'Username' and one for 'Passowrd'. At the bottom left is a 'Login' button, and at the bottom right is a link 'Don't have an account? Register.'

ログインフォームの送信

- 現在のログイン画面では、実質的な機能がありません。ログインボタンを押した際に入力されたフォームデータに対して、サーバがレスポンスするように望みます。
- そのため、まず、フォームが送信される HTTP リクエストの **パスとメソッド** を HTML コードに定義する必要があります：

```
<form action="/login" method="post">
```

- <form> タグの **action** 属性がリクエスト先の URL を定義し、**method** 属性がそのメソッドを定義しています。
- この例では、ログインボタンをクリックすると、「`http://localhost:8080/login`」に POST リクエストが送信されます。

フォームのパラメータ名を設定

- ユーザーがログインフォームに入力したデータをサーバに送信するために、`<input>` タグの `name` 属性でパラメータ名を設定する必要があります：

```
1 <div>
2   <label for="username">Username</label>
3   <input id="username" name="username" type="text"></input>
4 </div>
5
6 <div>
7   <label for="pwd">Passowrd</label>
8   <input id="pwd" name="password" type="password"></input>
9 </div>
```

コントローラでフォームデータを処理

- 次に、コントローラーで、このリクエストに応答するための以下のメソッドを記述する必要があります：

```
@PostMapping("/login")
public String login(@RequestParam String username, @RequestParam String password) {
```

次へ ➞

◀ 前へ

```
@PostMapping("/login")
public String login(@RequestParam String username, @RequestParam String password) {
```

- ここで、`@PostMapping("/login")` アノテーションは、このメソッドが「localhost:8080/login」に送信される POST リクエストを処理することを示します。（`<form>` の `action`、`method` 属性に対応。）
- `@RequestParam` アノテーションは、このリクエストが `username` と `password` の 2 つのパラメータを持つことを示します。（`<input>` の `name` 属性に対応。）

Note !

Java のパラメータ名は `<form>` の `name` 属性に対応すべき。`id` 属性ではありません！

コントローラで内容を返す

- ログインボタンをクリックした後、ユーザーに hello ページを表示させたいと考えています。localhost:8080/helloといった、hello ページに戻るための URL は既にできているので、ブラウザでこの URL に転送[Redirect]させるだけです。
- リダイレクトするためには、次のような文字列を返します：

```
1 @PostMapping("/login")
2 public String login(@RequestParam String username, @RequestParam String password) {
3     return "redirect:/hello";
4 }
```

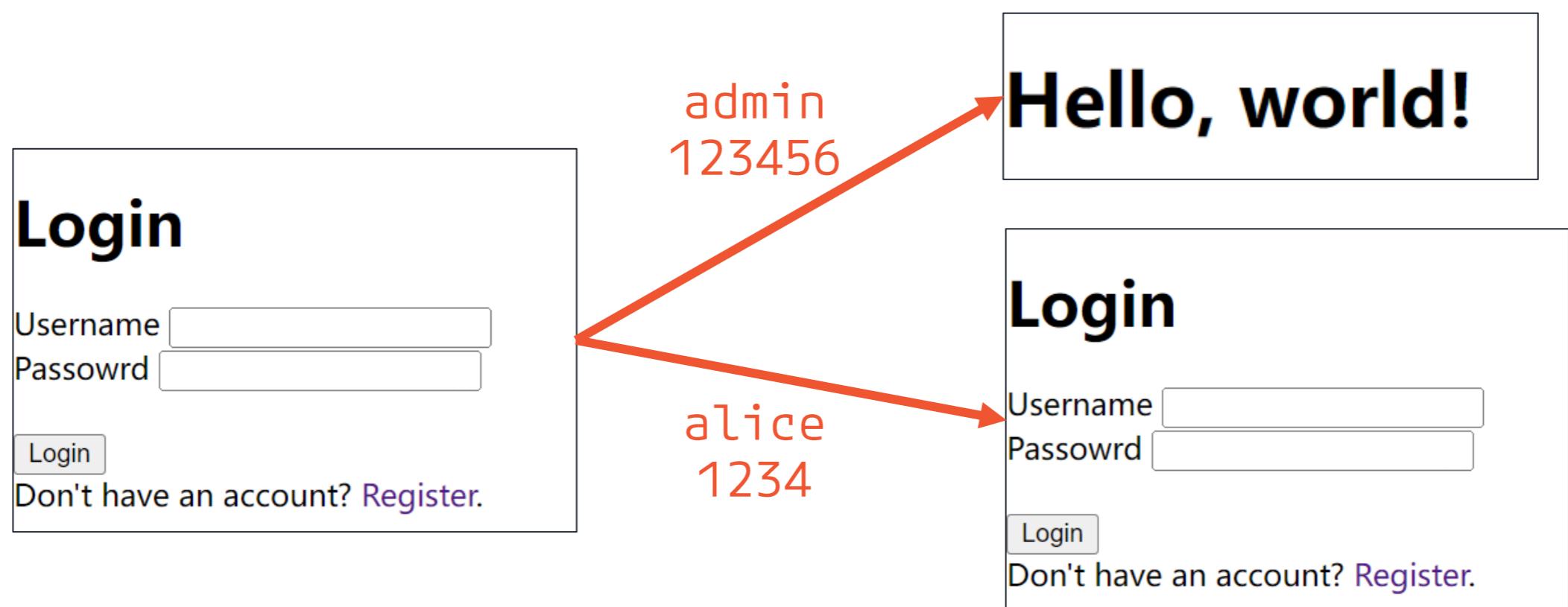
- こうすることで、ログインボタンをクリックすると、ユーザーは自動的に localhost:8080/hello に転送されて、最終的に hello ページが開きます。



Question and answer

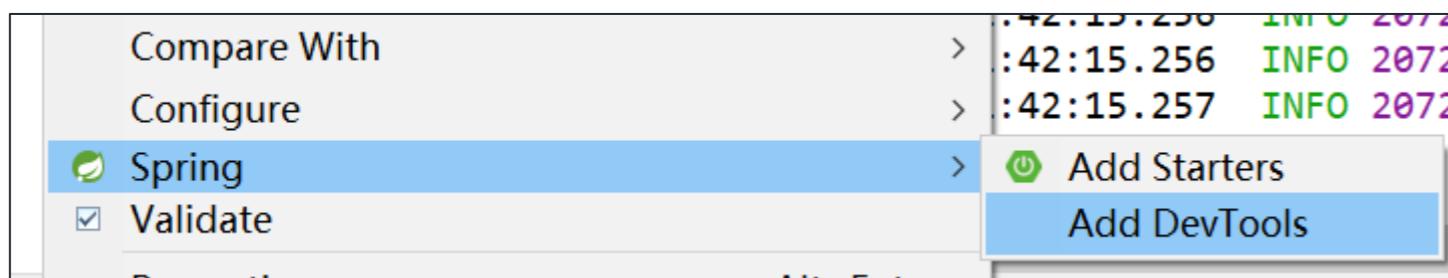
演習：入力の検証

- 次は、サーバーがユーザー入力を検証できるようにしたい。
- ユーザー名が admin、パスワードが 123456 の場合だけ、`hello` ページにリダイレクトされます。ユーザー名またはパスワードが間違っている場合は、ログインページにリダイレクトして、再入力を求めます。



DevTools

- コードを変更する毎にサーバーを手動で再起動する必要がありますが、**DevTools** プラグインを使用することでこのプロセスを簡略化することができます。
- プロジェクトを右クリック → Spring → Add DevTools:



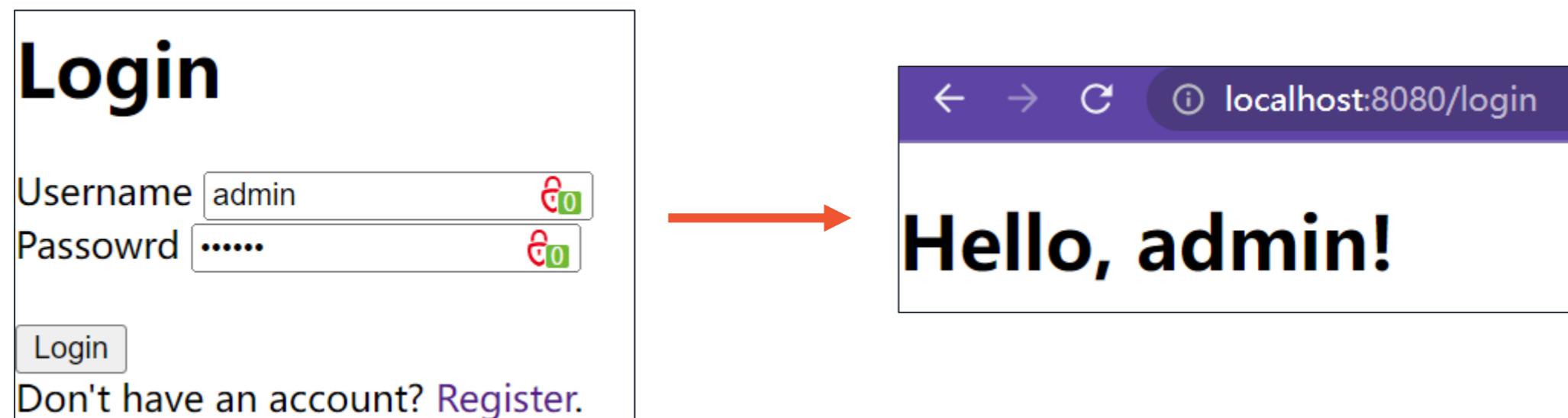
- プラグインをインストールすると、コードを保存するごとに Spring が自動的にサーバーを再起動します。ただし、コンピュータが重くなる恐れもあるので、状況に応じて使うこと。



- 1 プロジェクト作成
- 2 初めてのサーバ作成
- 3 Thymeleaf

動的ウェブページ

- 今まででは、変化のない静的ウェブページをブラウザに返すことしかできません。データ（フォームに入力したデータ、データベースのデータなど）をもとに、サーバーが動的なページを生成できるようにしたい。例えば、hello ページで、「Hello, [ユーザーの名前]！」と表示したい：



- つまり、前に (⬅ § 4.1.1) 紹介した動的ウェブページの概念を実現したいです。

テンプレートエンジン

- **テンプレートエンジン** [Template Engine] は、このような動的ウェブページを生成するために利用されています。
- テンプレートエンジンを利用する場合、専用の**テンプレート言語**（多くは HTML 言語の変形）を使ってページを作成できます。コントローラのメソッドから与えられたデータを取り、そのデータを使って最終的な HTML ページをレンダリングすることができます。
- もちろん、(Servlet の時のように) Java のコードでページを直接生成することも可能ですが、テンプレートエンジンを利用することによって、Java コード内の HTML コードの記述を減らし、HTML コードと Java コードを分離できます。コードの可読性を向上させ、開発効率を高められます。

Thymeleaf

- 従来の Java サーバ開発でよく使われていたテンプレートエンジンには、JSP などがあります。しかし、Spring Boot では、軽量でオープンソースなエンジンである **Thymeleaf** の使用が推奨されます。
- JSP の最大の問題は、サーバーから送られてくるデータを表示するために、いくつかの特殊な HTML タグ（例えば「<%@ variable%>」）を使用していることです。そのせいで、サーバ起動せずにブラウザで画面を確認することはできません。



Thymeleaf

次へ ➞

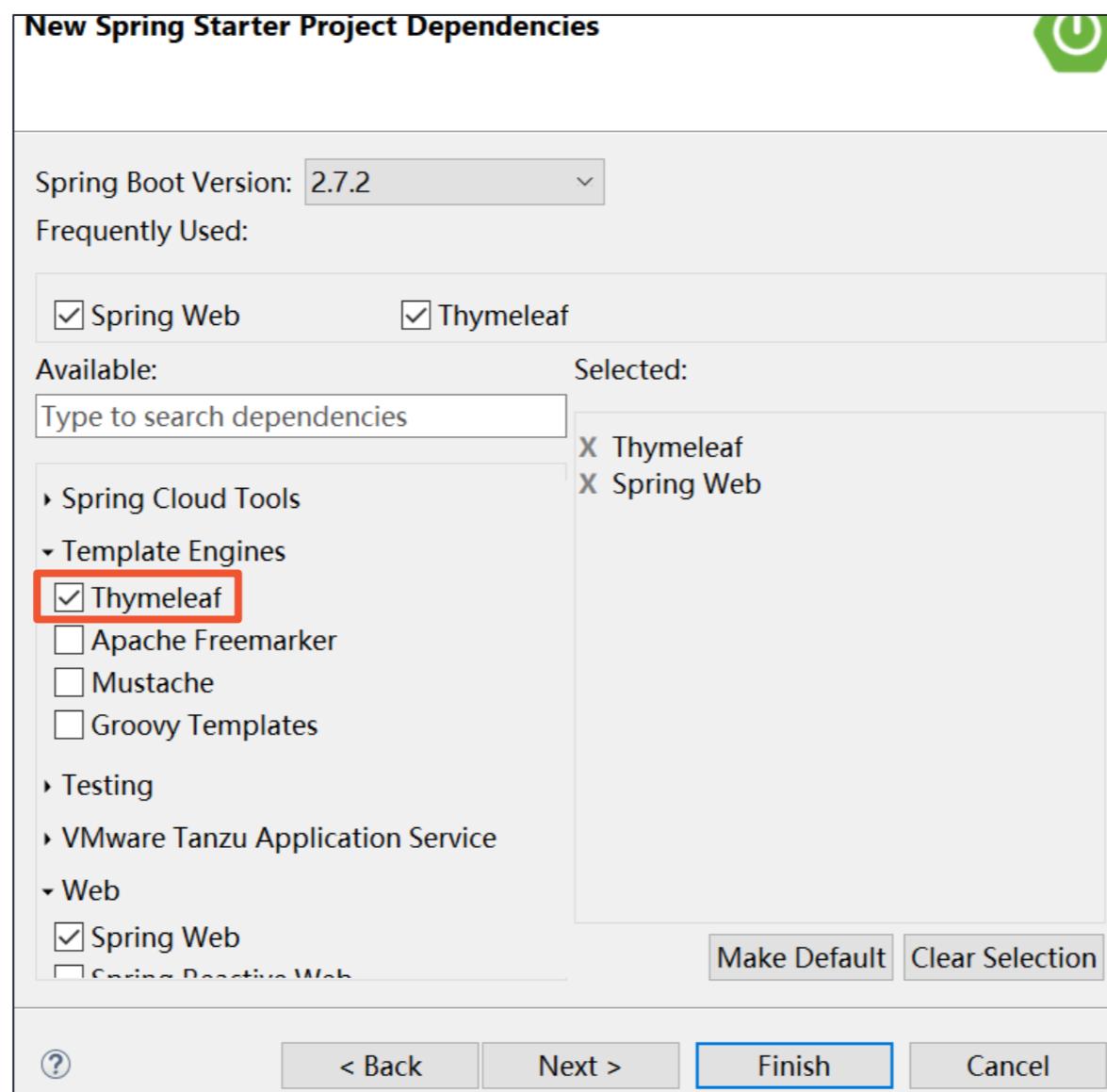
◀ 前へ

- これに対し、Thymeleaf が使用するテンプレート言語は、基本の HTML タグだけに基づいて効果を実現します。データを表示するためには、既存の HTML タグに**特殊な属性**を追加するだけです。これによって、サーバを起動しなくでも、ブラウザでこれらのページが正しく確認しテストすることができます。



Thymeleaf の依存関係の追加

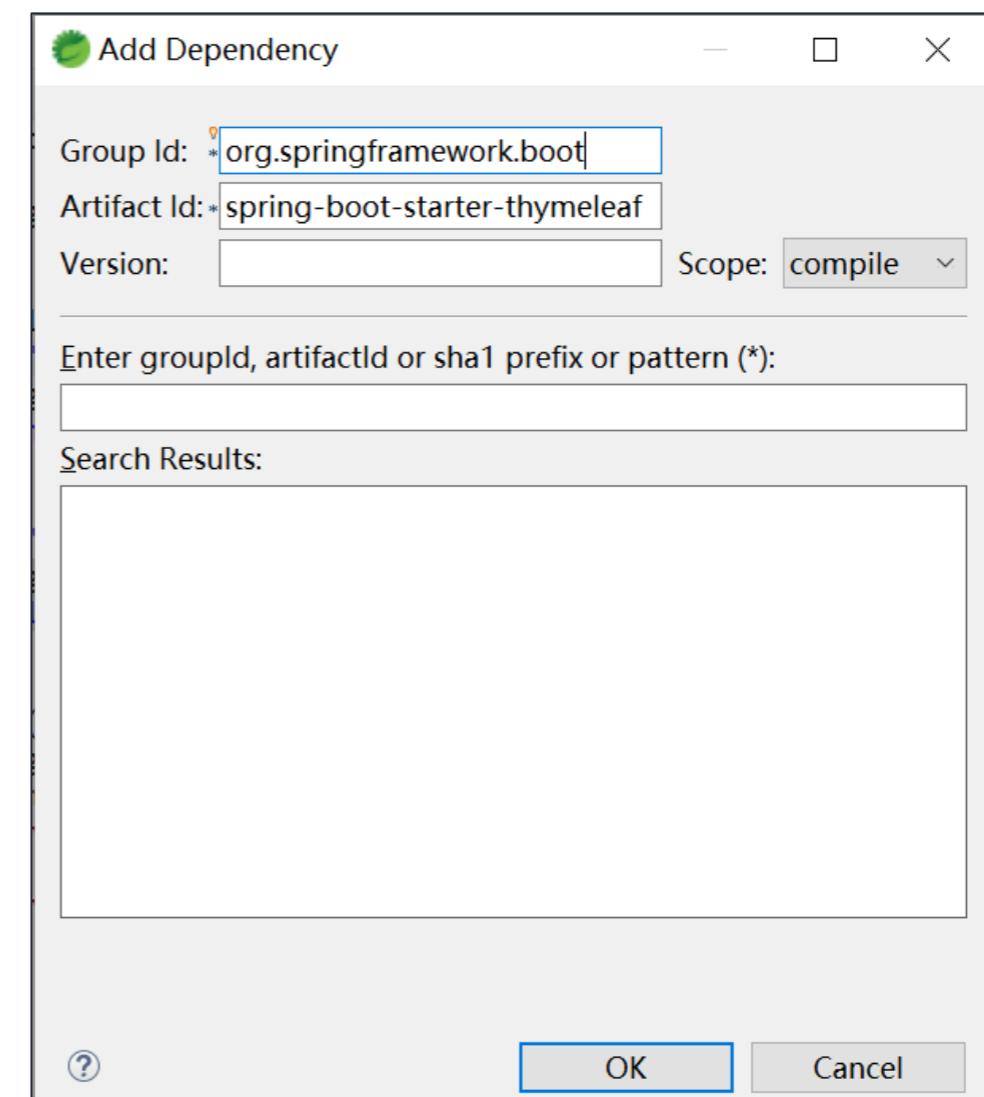
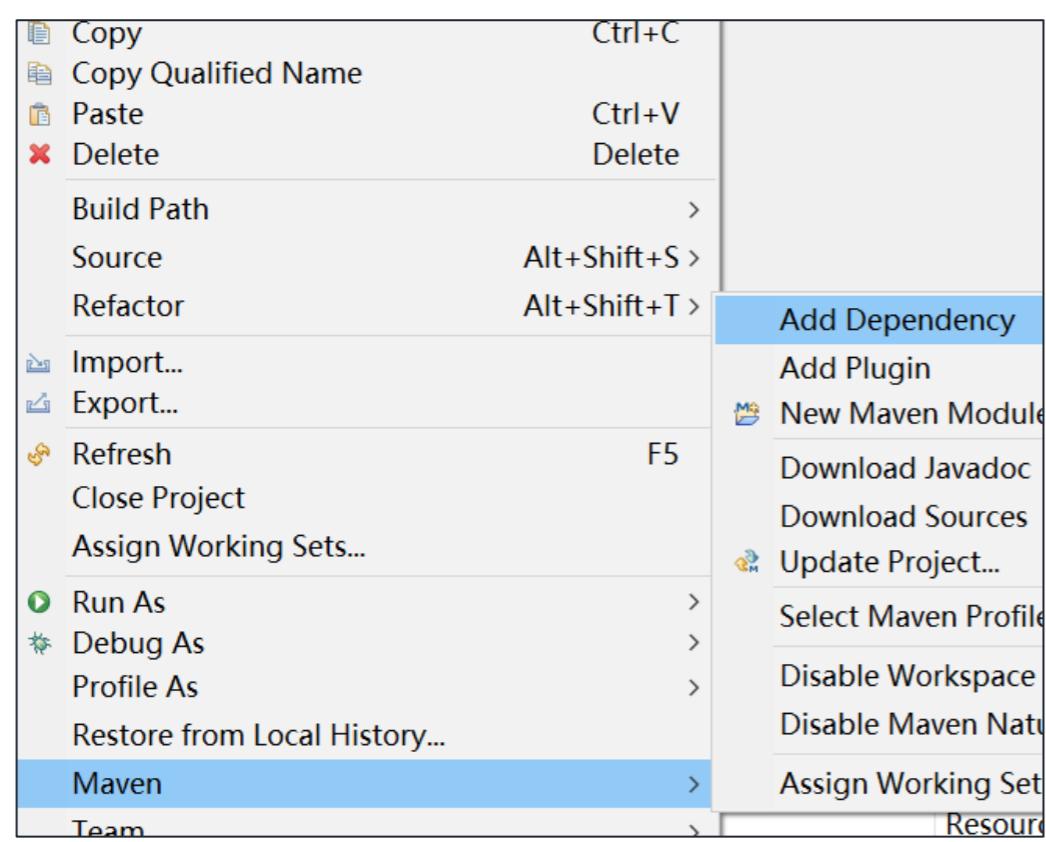
- Thymeleaf を使用するには、プロジェクトを作成するときに Thymeleaf を選択する必要があります：



次へ ➞

◀ 前へ

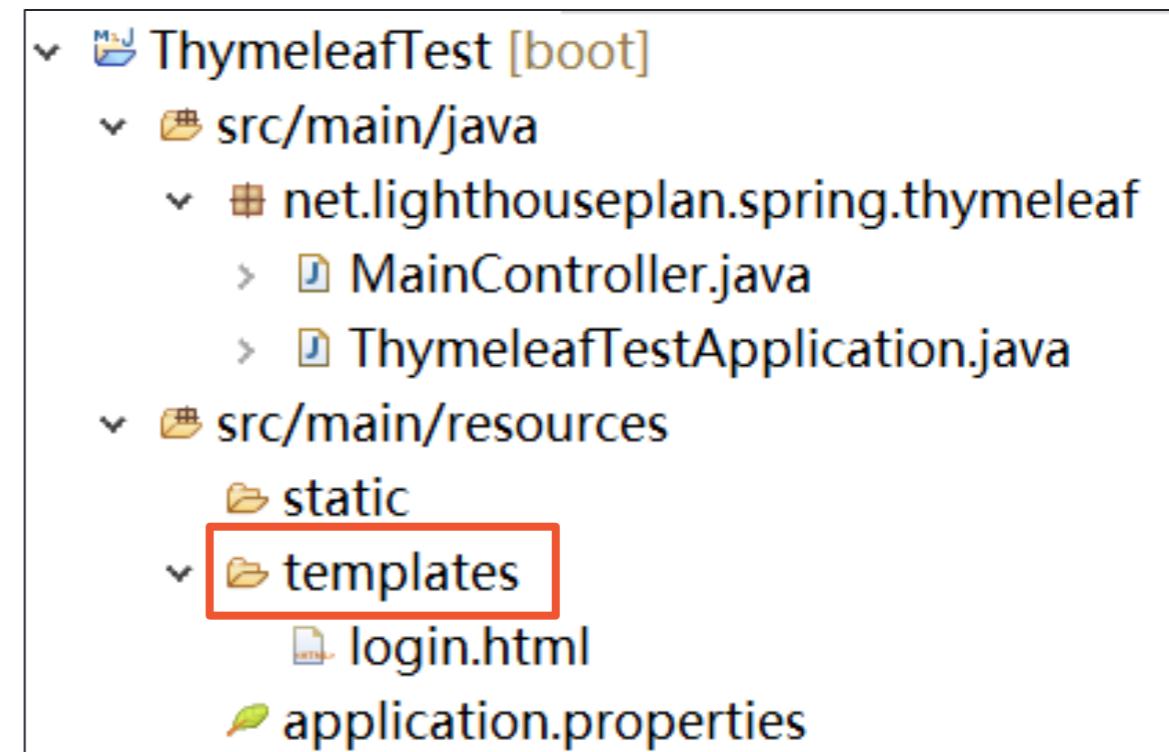
- プロジェクト作成後に依存関係を追加する場合、プロジェクトを右クリック → Maven → Add Dependency で、Thymeleaf の groupId と artifactId を入力します：



ページの追加

- まず、静的ログインウェブページを追加してみましょう。コントローラは先ほどと同じように書きます。
- 続いて、HTML ファイルですが、今回は、**templates** フォルダに入れます：

```
1 @Controller
2 public class MainController {
3     @GetMapping("/login")
4     public String getLoginPage() {
5         return "login.html";
6     }
7 }
```



Thymeleaf のページへ情報を渡す

- 次に、ユーザがログインした時の POST リクエスト用のコントローラを書きます。コントローラで、ログインに成功し、hello ページにアクセスする時、ユーザーに関する情報を Thymeleaf テンプレートに渡したいです。
- これを実現するために、2 つの方法があります：
 - Model オブジェクトを利用
 - ModelAndView オブジェクトを利用
- 書き方は違いますが、効果は同じです。自分で書く時にはどちらかを選んで書きなさい。

Model オブジェクトの使用

- 一つ目の方法は、Model オブジェクトの使用。
- 元のメソッドに新しい Model 型のパラメータを追加し、その addAttribute() メソッドでデータを追加します：

```
1 @PostMapping("/login")
2 public String login(@RequestParam String username,
3                      @RequestParam String password, Model model) {
4     model.addAttribute("name", username);
5     return "hello.html";
6 }
```

ModelAndView オブジェクトの使用

- もう一つ目の方法は、**ModelAndView** オブジェクトの使用です。
- ModelAndView 型のパラメータをメソッドに追加し、**addObject()** メソッドでデータを、**setViewName()** で HTML テンプレートのパスを設定します。最後に、**ModelAndView** オブジェクトを戻り値として返します：

```
1 @PostMapping("/login")
2 public ModelAndView login(@RequestParam String username,
3                           @RequestParam String password, ModelAndView mav) {
4     mav.addObject("name", username);
5     mav.setViewName("hello.html");
6     return mav;
7 }
```

テンプレートページの作成

- templates フォルダに先ほどと同じ hello.html ファイルを作成します。
- コントローラーから渡された情報をどうやってに取得しますか? Thymeleaf では、タグに特別な属性 **th:text** が追加てきて、タグの内容が指定されたデータのパラメータの値になるようになっています:

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

- この <h1> タグの値が、コントローラーから渡された「name」の値となります。「\${ }」中での内容は、OGNL 式という言語になります。簡単に言うと、Java コードのようなコード表現です。

次へ 

◀ 前へ

- hello.html のタグを変更して結果をチェックしましょう:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title>Hello Spring</title>
6 </head>
7
8 <body>
9   <h1>Hello, <span th:text="${name}"></span>!</h1>
10 </body>
11 </html>
```

Thymeleaf の機能

- Thymeleaf は、タグの内容を修正するなどの基本的な機能に加え、以下のような豊富なテンプレート機能を備えます：
 - HTML 属性の値の設定、
 - 基本的な演算子や、Java メソッドなどの使用、
 - リスト、オブジェクトのデータ表示、
 - 条件の判断、
 - URL の動的処理など。
- これらの機能の詳細は、公式ドキュメントを参照：
 <https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html>

HTML 属性を設定

- text 属性の設定だけでなく、他の HTML 属性も、元の属性名の前に「**th:**」を付けるだけで設定できます。例えば、以下のコードでは、タグの class 属性を「type」というデータで設定します：

```
<h1 th:class="${type}">Hello</h1>
```

インライン式

- th:text 属性を使うほか、テキストの中に直接「**[[]]**」という表記でインライン式を記述することもできます：

```
<h1>Hello, [[${name}]]!</h1>
```

- 上記のコードの **[[\${name}]]** の部分は、name の値としてレンダリングされます。この書き方は、大量のテキストの中の特定の部分だけを変更したい場合に便利です。

演算子の使用

- Java の基本的な**演算子**は Thymeleaf で直接使用することができます。例えば、**算術演算子**:

```
<p th:text="${num1 * num2 / 3}"></p>
```

- 三項演算子**:

```
<p th:text="${num1 > num2 ? 0 : 1}"></p>
```

- ただし、いくつかの相違点に注意すること:

- 文字列はシングルクオート「'」で表現:

```
<p th:text="'num1 = ' + num1 + '.'"></p>
```

- 論理演算子は「&&」「||」ではなく、「and」「or」を使用。

条件判断

- **th:if** 属性は、ブール値の式を評価し、式が真である場合にのみ要素を表示します。
- 例えば、以下のタグは、受信した num というデータが 60 以上の場合のみ表示されます：

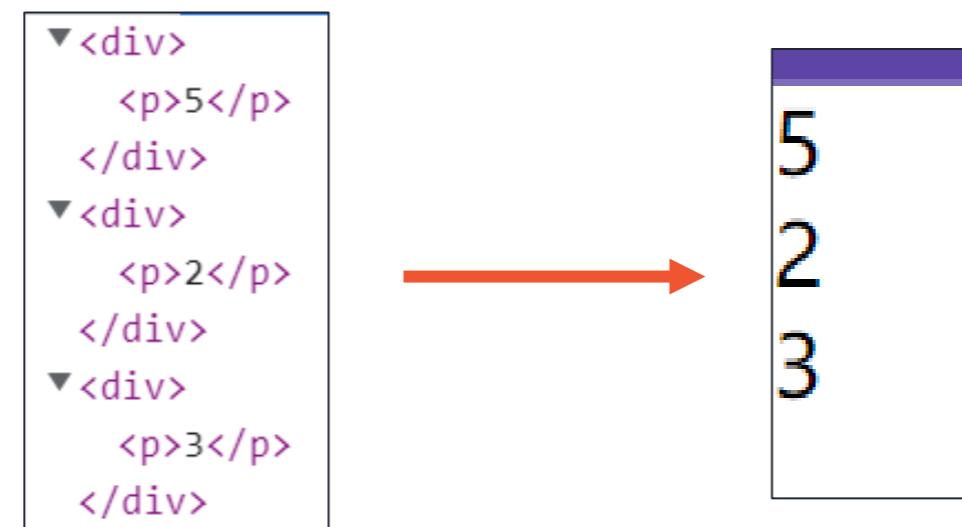
```
<p th:if="${num > 60}">num > 60!</p>
```

リストのトラバース

- **th:each** を使って、配列、リスト、コレクションなどを順番に列挙して表現できます：

```
1 <div th:each="num : ${nums}">
2   <p th:text="${num}"></p>
3 </div>
```

- ここで、`nums` は `List.of(5, 2, 3)` のような、リストデータです。リスト内にある `num` ごとに、`<div>` が自動的に生成します：



オブジェクト

- 渡されたパラメータがオブジェクトである場合、Java のように「.」でプロパティを取得し、メソッドを呼び出すことができます：

```
<p th:text="${student.name}"></p>
<p th:text="${student.sayHello()}"></p>
```

オブジェクト式

- しかし、同じオブジェクトの多くのプロパティを使用する場合、冗長になりがちです：

```
1 <div>
2   <p th:text="'Name: ' + ${studentInformation.name}"></p>
3   <p th:text="'Age: ' + ${studentInformation.age}"></p>
4   <p th:text="'Score: ' + ${studentInformation.score}"></p>
5 </div>
```

- th:object 属性**と「***{ }** 」を使って簡潔に表現できます：

```
1 <div th:object="${studentInformation}">
2   <p th:text="'Name: ' + *{name}"></p>
3   <p th:text="'Age: ' + *{age}"></p>
4   <p th:text="'Score: ' + *{score}"></p>
5 </div>
```

URL 处理

- テンプレートによる生成のため、`<link>` や `<a>` の `href` 属性や、`<script>` の `src` 属性など、サーバ上のファイルの相対パス（URL）の計算が煩雑です。
- URL を「`@{}`」で囲むと、Thymeleaf が自動的にサーバ上においての正しい URL を計算します。
- 例えば、次のコードは `static/css/main.css` ファイルをインポートします：

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

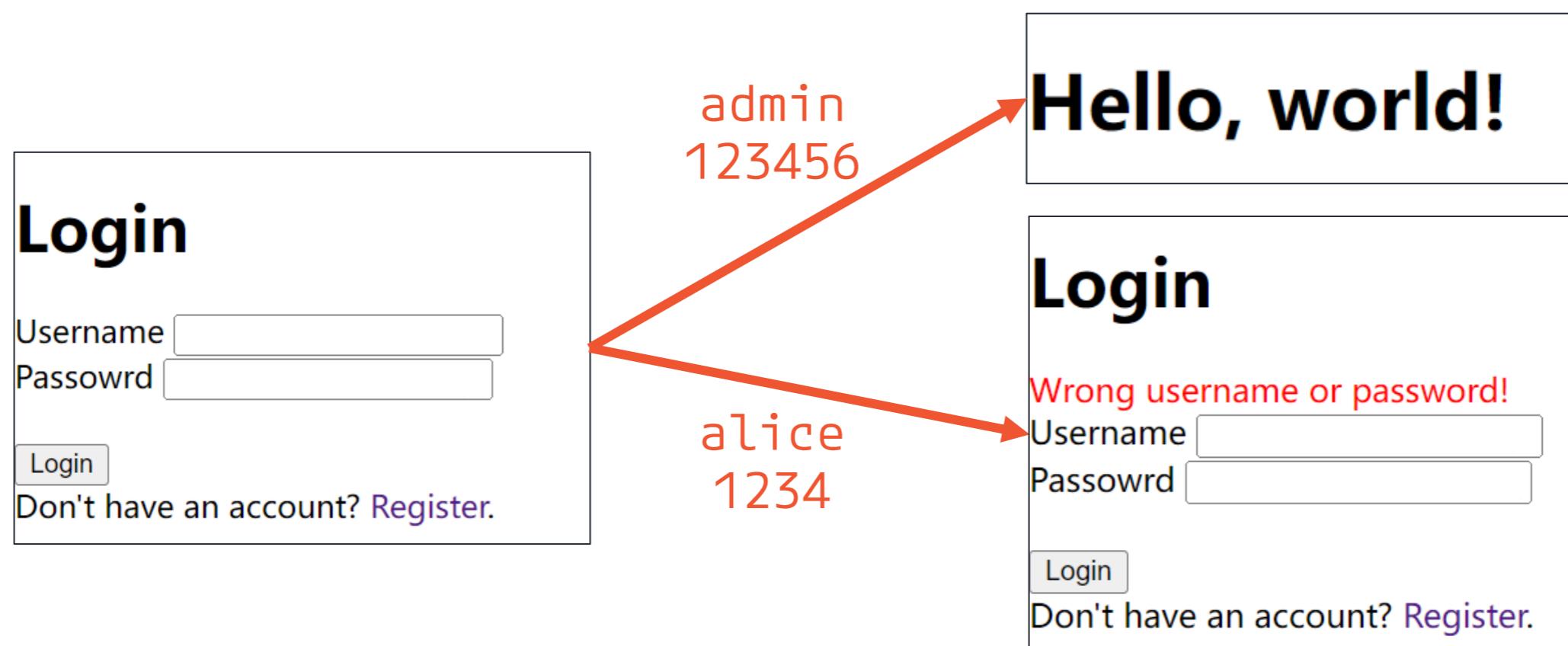
- これで、他の静的リソースファイル（CSS、JavaScript、画像など）を今まで通り `static` フォルダの下に置いておくことができます。



Question and answer

演習： エラーメッセージの表示

- ユーザーが間違ったユーザー名とパスワードを入力した場合、ログインページに戻るだけでなく、追加のエラーメッセージも表示させたい。
- 新しいページを作成せずに、Thymeleaf でこの機能を実現できますか。



まとめ

Sum Up

1. Spring Boot プロジェクトの作成と Maven の依存パッケージを追加する方法。
2. Spring プロジェクトでのコントローラーの書き方: @Controller、@XxxMapping、@RequestParam などのアノテーションの意味。
3. Spring プロジェクトで動的ウェブページを生成する方法: Thymeleaf テンプレート言語の基本文法。

THANK YOU!