Thymeleaf

1. Introduction to Thymeleaf
2. Thymeleaf syntax

1. Introduction to Thymeleaf

- What is Thymeleaf?
- Thymeleaf sample application
- Configuring Maven dependencies
- Setting Thymeleaf as the view resolver
- Defining template pages

What is Thymeleaf?

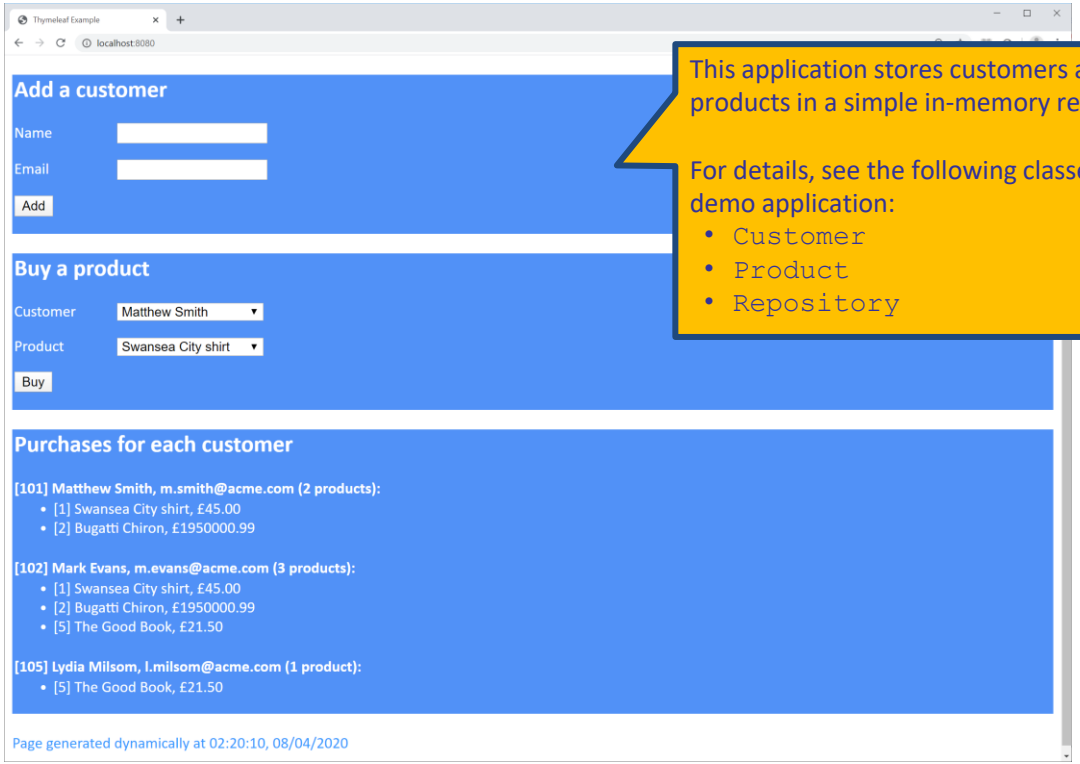
- Thymeleaf is an open-source view template framework for HTML5, XHTML, or XML resources
  - A popular alternative to JSP, especially when you have a Spring Boot app with an embedded servlet container

- Benefits:
  - Clean and powerful templating syntax
  - Allows for faster and offline prototyping of pages, because you can view pages outside a Web server

- Note:
  - You can use Thymeleaf in Spring and non-Spring apps

Thymeleaf Sample Application

- We'll explore `demo-thymeleaf` in this chapter



Configuring Maven Dependencies

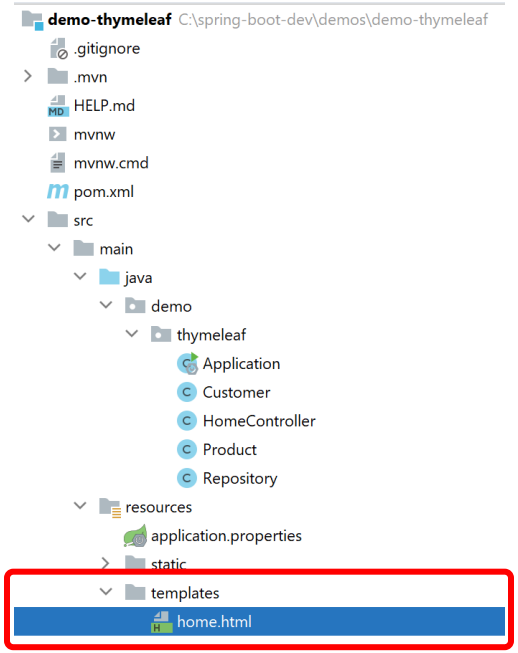
- In Spring Boot, you just need to add the Spring Boot Starter for Thymeleaf
  - Auto-configures all the beans needed by Thymeleaf

```
<dependencies>

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

</dependencies>
```

pom.xml in demo project

Defining Template Pages (1 of 2)

- You put your template pages in the folder specified by the view resolver
  - By default, this is the `templates` folder



Defining Template Pages (2 of 2)

- You define template pages as follows
  - The `<!DOCTYPE>` declaration is necessary, to prevent errors
  - The `xmlns:th` namespace declaration allows you to use Thymeleaf attributes in your HTML element, e.g. `th:field`

```
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-spring4-4.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org">
<head>
   …
</head>
<body>
   …
</body>
</html>
```

/templates/home.html

2. Thymeleaf Syntax

- Online vs. offline operation
- Accessing model objects
- Accessing properties on an object
- Creating mixed content
- Binding fields to form values
- Creating URLs
- Creating repeated items
- Conditional rendering
- Accessing bean values
- Linking to resources

Online vs. Offline Operation

- Thymeleaf can replace text in an HTML page

```
<select name="customerID">
    <option th:text="${customer.name}" … > Customer name [offline] </option>
</select>
```

/templates/home.html

- If the page is loaded offline (without a Web server), the "normal" text is displayed
  - In this example: `Customer name [offline]`

- If the page is accessed via a Web server, the text will be replaced with whatever's in `customer.name`

Accessing Model Objects (1 of 2)

- Controllers can populate the model with useful values

```
@Controller
public class HomeController {

  @RequestMapping(value = "/")
  public String home(Model model) {
    model.addAttribute("newCustomer", new Customer());
    model.addAttribute("Customers", Repository.getAllCustomers());
    model.addAttribute("Products", Repository.getAllProducts());
    return "home";
  }
  …
}
```

HomeController.java

Accessing Model Objects (2 of 2)

- You can access model values in a template page
  - Use the syntax `${nameOfModelAttribute}`

- Example in our template page `home.html`:

```
<form action="#" th:object="${newCustomer}" method="post" … >
```

```
<select name="customerID">
    <option th:each="c : ${Customers}"
            th:value="${c.customerID}"
            th:text="${c.name}">Customer name [offline]</option>
</select>
```

```
<select name="productID">
    <option th:each="p: ${Products}"
            th:value="${p.productID}"
            th:text="${p.description}">Product description [offline]</option>
</select>
```

/templates/home.html

Accessing Properties on an Object

- The `${…}` syntax also allows you to access a specific property on an object
  - Thymeleaf uses OGNL syntax
  - E.g. `${c.customerID}` calls `getCustomerID()`

- Example:

```
<select name="customerID">
    <option th:each="c : ${Customers}"
            th:value="${c.customerID}"
            th:text="${c.name}">Customer name [offline]</option>
</select>
```

```
<select name="productID">
    <option th:each="p: ${Products}"
            th:value="${p.productID}"
            th:text="${p.description}">Product description [offline]</option>
</select>
```

Creating Mixed Content

- If you want to mix static and dynamic text, you can use simple string concatenation

```
<div th:text="${'Hi' + newCustomer.name + ', how are you?'}"></p>
```

- If you want to mix static and dynamic HTML elements, the easiest way is to define nested elements

```
<div>Your name is <b><span th:text="${newCustomer.name}"></span></b></div>
```

Binding Fields to Form Values

- The `*{…}` syntax allows you to bind a field to a form value
- Example:
  - The input fields here are bound to the `name` and `emailAddress` properties in the `newCustomer` object

```
<form th:action="@{/addCustomer}" th:object="${newCustomer}" … >

    <p>
        <label>Name</label>
        <input type="text" th:field="*{name}" />
    </p>

    <p>
        <label>Email</label>
        <input type="email" th:field="*{emailAddress}" />
    </p>

    <p>
        <button type="submit">Add</button>
    </p>

</form>
```

/templates/home.html

Creating URLs

- The `@{…}` syntax allows you to create a URL
- Example:
  - The following form posts to the URL `/addCustomer`
  - This is mapped to `handleAddCustomer()` in the controller

```
<form th:action="@{/addCustomer}" th:object="${newCustomer}" … >

    <p>
        <label>Name</label>
        <input type="text" th:field="*{name}" />
    </p>

    <p>
        <label>Email</label>
        <input type="email" th:field="*{emailAddress}" />
    </p>

    <p>
        <button type="submit">Add</button>
    </p>

</form>
```

/templates/home.html

Creating Repeating Items

- Use `th:each` to create repeated items
  - Useful for iterating through collections, in a similar way to using the JSTL tag library

- Example:
  - The following `<option>` uses `th:each` to iterate over customers
  - Thymeleaf will create a separate `<option>` for each customer

```
<select name="customerID">
    <option th:each="c : ${Customers}"
            th:value="${c.customerID}"
            th:text="${c.name}">Customer name [offline]</option>
</select>
```

/templates/home.html

Conditional Rendering

- Use `th:if` for conditional rendering
  - Useful for hiding/displaying items based on current data

- Example:
  - The following `<div>` displays customer details, but only if the customer has purchased some products

```
<div th:if="${!customer.products.isEmpty()}">
    <h4 th:text="${customer} + ':' ">Customer details [offline]</h4>
    <ul>
        <li th:each="product : ${customer.products}" th:text="${product}"> … </li>
    </ul>
</div>
```

/templates/home.html

Accessing Bean Values

- You can access bean values directly
  - Here's a bean value:

```
@SpringBootApplication
public class Application {
    …

    @Bean
    @Scope("request")
    public String timestamp() {
        // Return a string containing the current date and time…
    }
}
```

Application.java

  - Here's how to access it in the template page:

```
<footer>
    <p th:text="${@timestamp}">Timestamp</p>
</footer>
```

/templates/home.html

Linking to Resources

- You can define stylesheets and other static resources
  - Put these files in the normal place for static content in a Spring Boot app, e.g. in the `/static` folder

```
<head>
    <title>Thymeleaf Example</title>
    <link th:href="@{/styles.css}" type="text/css" rel="stylesheet" />
</head>
```

/static/styles.css

Summary

- Introduction to Thymeleaf
- Thymeleaf syntax