



AIF233201 - Pemrograman Berbasis Web Rendering & Request

by Raymond Chandra Putra
Pascal Alfadian Nugroho
Keenan Adiwijaya Leman
31-10-2024



INFORMATIKA
UNPAR

Overview

- Thymeleaf
- Handling Request
 - GET Request
 - POST Request
- Project Structure



Rendering



INFORMATIKA
UNPAR

Dynamic SSR

- In web application, usually we deal with dynamic content. So dynamic SSR is more used than static SSR.
- To use dynamic SSR we use templating engine to modify document before send it to client side.
- One of common templating engine used in Java Spring Boot called Thymeleaf

Thymeleaf

- Thymeleaf is a modern server-side Java template engine for both web and standalone environments, capable of processing HTML, XML, JavaScript, CSS and even plain text.
- The main goal of Thymeleaf is to provide an elegant and highly-maintainable way of creating templates.
- To achieve this, it builds on the concept of Natural Templates to inject its logic into template files in a way that doesn't affect the template from being used as a design prototype. This improves communication of design and bridges the gap between design and development teams.
- Thymeleaf has also been designed from the beginning with Web Standards in mind – especially HTML5 – allowing you to create fully validating templates if that is a need for you.



How to use Thymeleaf

- Thymeleaf use attributes in `org.springframework.ui.Model` as source of data in templating.
- If we use class annotation `@Controller`, when we return string (without `@ResponseBody`), Spring will find corresponding template in folder `resource/template`.

```
@GetMapping("/")  
public String index(Model model) {  
    model.addAttribute("name", "John Doe");  
    return "index";  
}
```

How to use Thymeleaf (2)

- Thymeleaf use key value pair for the attributes.
- Then, in template HTML file, we use the name of the attribute where dynamic content used.

```
<h1>Hello, <span th:text="${name}">World</span>!</h1>  
<p>Text <span data-th-text="${name}"></span> Other Text</p>
```

- There are 2 alternatives of using thymeleaf:
 - th:*
 - data-th-*
 - The th:* is not really valid HTML5 document, and sometimes IDE complaining about this.

Escaped Text

- When we insert html text, thymeleaf automatically escape the text. To use unescaped text, we use `utext`
- `Htmldata` : "`bold`"

```
<p>Text 1 <span data-th-text="${htmldata}"></span> Other Text</p>
```

```
<p>Text 2 <span th:utext="${htmldata}"></span> Other Text</p>
```

- Output:

Text 1 `bold` Other Text

Text 2 **bold** Other Text

Attribute Template

- Other than text content in tag, we also can use dynamic content in attributes.
- We can use 2 alternatives that have the same results:

```
<input type="text" name="name" th:attr="value=${name}" />  
<input type="text" name="name" th:value="${name}" />
```

- “value” can be changed for all attributes used in HTML.

Templating in Javascript

- We can also use thymeleaf in javascript as following:

```
<script th:inline="javascript">  
    let text = /*[[${name}]]*/ "Default Text";  
    console.log(text);  
</script>
```

- Thymeleaf will use default text if file opened manually because of commenting.
- But automatically replace with dynamic content if templating used.

Branching

- What if we want to show different kinds of content depends on the state of the variable?
- Let's explore these codes:

```
<div th:if="${isAuth}">Hello, <span th:text="${name}">World</span></div>  
<div th:if="${isAuth} == false">Please login</div>  
<div th:class="${isAuth} ? yes : no">Test</div>
```

Looping

- When we want to use repeating content, looping can be used.
- For examples:

```
<ul>  
  <li th:each="item : ${arrAngka}"><span th:text="${item}"></span></li>  
</ul>
```


Request



INFORMATIKA
UNPAR

Request

- In previous slides we had already showed the basic of HTTP request.
- For each request, usually we also send the data inside the requests. Data send in request can be via:
 - URL
 - Request body

Handling GET Request

- RequestParam vs PathVariable

```
@GetMapping("/hello")
public String hello(@RequestParam String name) {
    return "Hello " + name;
}
```

```
@GetMapping("/hello2/{name}")
public String hello2(@PathVariable String name) {
    return "Hello " + name;
}
```



Handling GET Request (2)

- Instead of full
 - `@RequestParam(name = "name", required = false, defaultValue = "World")`
- We can use default version of `@RequestParam`
 - Parameter variable name automatically used as name
 - Required value as default
 - If we want to customize, we have to use parameterized version

POST Request

- When POST Request used, data are inside request body instead of URL.
- There are several alternative to access request body:
 - Request param
 - HttpServletRequest
 - Model Attribute

Handling POST Request

- RequestParam

```
@PostMapping("/persons")
@ResponseBody
public String createPerson(
    @RequestParam String fname,
    @RequestParam String lname
) {
    return fname+" "+lname;
}
```



Handling POST Request (2)

- HttpServletRequest

```
@PostMapping("/persons2")
@ResponseBody
public String createPerson2(HttpServletRequest request) {
    String fname = request.getParameter("fname");
    String lname = request.getParameter("lname");
    return fname+" "+lname;
}
```



Handling POST Request (3)

- Model Attribute

```
@PostMapping("/persons3")
@ResponseBody
public String createPerson3(@ModelAttribute Person person) {
    String fname = person.getFname();
    String lname = person.getLname();
    return fname+" "+lname;
}
```



Converter

- When handling inputs, sometimes we need to adjust format with other types according what we want to use in controller.
- For example:
 - String Date → LocalDate
 - String → Enum
 - Id → Database Object
 - Etc.

Create Custom Converter

```
8 public class StringToLocalDateConverter implements Converter<String, LocalDate> {
9     private static final DateTimeFormatter formatter =
10         DateTimeFormatter.ofPattern("yyyy-MM-dd");
11     @Override
12     public LocalDate convert(String source) {
13         System.out.println("Converting: " + source);
14         if (source == null || source.isEmpty()) {
15             return null;
16         }
17         return LocalDate.parse(source, formatter);
18     }
19 }
```



Registering Converter

```
1 package com.example.pbw.config;
2
3 import org.springframework.context.annotation.Configuration;
4 import org.springframework.format.FormatterRegistry;
5 import org.springframework.web.servlet.config.annotation.WebMvcConfigurer
6
7 import com.example.pbw.converter.StringToLocalDateConverter;
8
9 @Configuration
10 public class WebConfig implements WebMvcConfigurer{
11     @Override
12     public void addFormatters(FormatterRegistry registry) {
13         registry.addConverter(new StringToLocalDateConverter());
14     }
15 }
```



Handling Multipart Type

- When using file in forms, for example uploading file, we have to use multipart/formdata type.

```
<form method="post" action="/upload" enctype="multipart/form-data">  
  <input type="file" name="file" required>  
  <button type="submit">Upload</button>  
</form>
```

- In server we receive the file object using MultipartFile type (`org.springframework.web.multipart.MultipartFile`)
- But, after server receive the files, we handle it like how we process File in Java (input stream, files, java nio, etc)

Handling Uploaded File

```
50 @PostMapping("/upload")
51 public ResponseEntity<String> uploadFile(@RequestParam MultipartFile file) {
52     if (file.isEmpty()) {
53         return ResponseEntity.badRequest().body("No file uploaded");
54     }
55
56     try {
57         //[asumsi: folder uploads sudah dibuat]
58
59         String filename = UUID.randomUUID().toString() + "_" + file.getOriginalFilename();
60         Path destFile = Paths.get(uploadDir).resolve(filename);
61         Files.copy(file.getInputStream(), destFile);
62
63         return ResponseEntity.ok("File uploaded successfully: " + destFile.getAbsolutePath());
64     } catch (IOException e) {
65         return ResponseEntity.internalServerError().body("Failed to upload file: " + e.getMessage());
66     }
67 }
```

A decorative graphic on the left side of the slide, consisting of a grid of blue squares of varying shades (light blue, medium blue, dark blue) that curves and tapers towards the top right corner.

Extra: Architecture



INFORMATIKA
UNPAR

Project Structure

- Layered Architecture vs Feature-Based Architecture

```

✓ structure1
  > common
  ✓ controller
    J AccountController.java
    J UserController.java
  ✓ model
    J Account.java
    J User.java
  ✓ repository
    J AccountRepository.java
    J UserRepository.java

```

```

✓ structure2
  ✓ account
    J Account.java
    J AccountController.java
    J AccountRepository.java
  > common
  ✓ user
    J User.java
    J UserController.java
    J UserRepository.java

```



Component

- `@Component` annotation is used to register class to Spring Bean and will be scanned using `ComponentScan` when we start the server.
- There are other annotation derived from component:
 - `@Controller`, `@RestController`, `@Service`, `@Repository`

Component (2)

- **@Controller**
 - Marks a class as a **web controller** in Spring MVC, responsible for handling web requests.
- **@RestController**
 - Marks a class as a **RESTful web service controller**.
 - @Controller but all method use @ResponseBody
- **@Service**
 - Annotate classes that perform service tasks or hold business logic. This helps separate business logic from controllers.
- **@Repository**
 - It serves as an abstraction layer between the application and the database, allowing for easy database CRUD (Create, Read, Update, Delete) operations.



Component Summary

Annotation	Purpose	Layer	Return Type
@Controller	Handles web requests and returns views	Presentation	Typically returns view names
@RestController	Handles RESTful web requests	Presentation	Returns data directly (JSON, XML)
@Service	Contains business logic	Business Logic	No specific return type; holds service methods
@Repository	Manages data access	Data Access	Returns entities or interacts with the database



Request Flow

1. **Controller:** Receives the request from the client, handles input validation, and delegates the business logic to the service layer.
2. **Service:** Applies business logic and coordinates interactions with one or more repositories to retrieve, update, or delete data as required.
3. **Repository:** Executes data access operations (such as fetching data from the database) and returns data to the service layer.
4. **Service:** Receives the data from the repository, applies any final business rules if necessary, and sends it back to the controller.
5. **Controller:** Packages the data as a response (like JSON or XML) and sends it back to the client.



Lombok

- Package Lombok is used to simplified POJO (Plain Old Java Object)
- Usually, POJO only have attributes, constructor, getter, setter.
- When we use Lombok, we reduce the boilerplate of adding getters, setters, constructor, etc.
- To use them, we simply add following dependencies:

```
compileOnly("org.projectlombok:lombok:1.18.26")  
annotationProcessor("org.projectlombok:lombok:1.18.26")
```



Lombok Data

- In model object, we simply use @Data annotation.

```
import lombok.Data;

@Data
public class Person {
    private String fname;
    private String lname;
    private LocalDate dob;
}
```



A decorative graphic on the left side of the slide, consisting of a grid of blue squares of varying shades (light blue, medium blue, and dark blue) that curves and tapers towards the top right corner.

Thanks



INFORMATIKA
UNPAR



**INFORMATIKA
UNPAR**

informatika.unpar.ac.id

informatika@unpar.ac.id

[if.unpar](https://www.facebook.com/if.unpar)

[if.unpar](https://www.instagram.com/if.unpar)