📒 Introduction To Thymeleaf

Thymeleaf is a template engine in Spring used to create dynamic HTML pages.
It allows you to insert values from your Java code (controller/model) directly into an HTML file.

- Normally, HTML is static.
- With Thymeleaf, you can mix HTML + Java data together.
- The controller sends data to the view (Thymeleaf template), and Thymeleaf replaces placeholders with real values.
- It makes your web pages dynamic.

Why use Thymeleaf?

- Works directly with HTML (you can open it in a browser even without a server).
- Easy to integrate with Spring Boot.
- Supports dynamic data rendering.
- Rich features: conditions, loops, fragments, internationalization, etc.

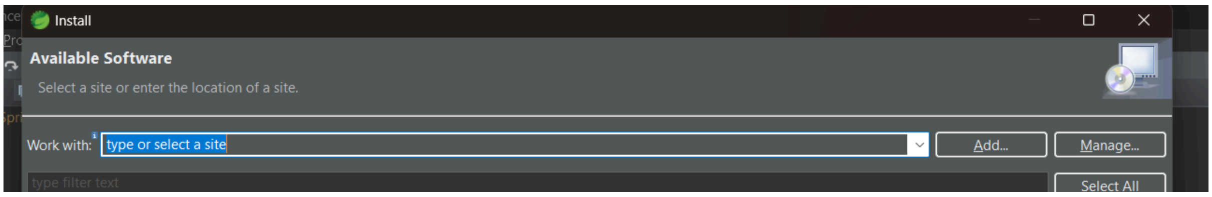
Thymeleaf	**JSP (Java Server Pages)**
Modern server-side **template engine** for generating dynamic HTML	Older **Java-based view technology** for creating dynamic web pages.
`.html` files (valid HTML – can be opened directly in browser).	`.jsp` files (not pure HTML – requires server to render).
Built-in and preferred template engine in Spring Boot.	Works with Spring MVC but **not default in Spring Boot**.
Easy, syntax looks like normal HTML with attributes (`th:text`, `th:if`).	More Java-style code inside HTML (`<% %>`, `${}` with JSTL).
Clean separation → No Java code in HTML, only expressions.	Can mix Java code and HTML (hard to maintain).
Supports expressions, conditionals, loops, fragments, layouts, i18n.	Limited, mainly relies on JSTL/EL for logic.
Since files are pure HTML, designers can open them directly without backend.	JSP cannot be previewed directly, needs server to render.

📒 Set-Up

Add following plugin in eclipse for better experience

1] Thymeleaf plug in

- Go to Help >> Install New Software
- Click Add



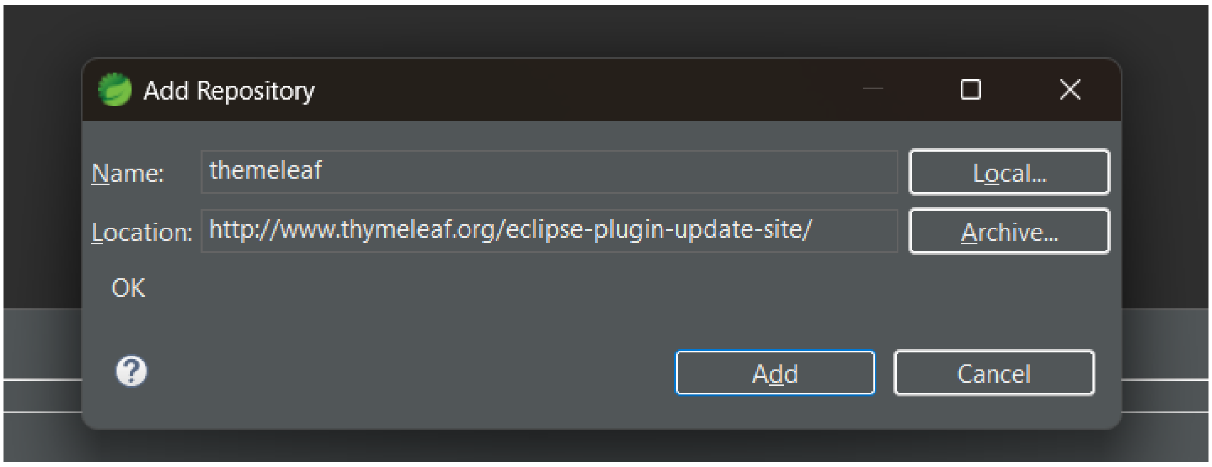
Install

Available Software

Select a site or enter the location of a site.

Work with: type or select a site

- then enter this update site URL (http://www.thymeleaf.org/eclipse-plugin-update-site/).



Add Repository

Name: themeleaf

Location: http://www.thymeleaf.org/eclipse-plugin-update-site/

OK

- Then follow the instructions by IDE and install the plugin.

📒 First Project

1. Create a Controller

```
@Controller
public class TestController {
        @GetMapping("/home")
        public String home(Model model) {

                model.addAttribute("name", "Gaurav");
                model.addAttribute("age", "21");
                return "index";
        }
}
```

- `@Controller` → Marks this as a Spring MVC controller.
- `Model` → Used to send data from backend to frontend.
- `return "index";` → Looks for `index.html` inside `src/main/resources/templates/`.

2. Enable Thymeleaf in HTML

Add this line inside `<html>` tag:

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

3. HTML File

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
	<h1>Thymeleaf</h1>
	<h2>My name is <span th:text="${name}"></span></h2>
	<h2>My age is <span th:text="${age}"></span></h2>
</body>
</html>
```

1. `xmlns:th="http://www.thymeleaf.org"`

- This line is added inside the `<html>` tag.
- It declares the **Thymeleaf XML namespace** (`th:`).
- It tells the browser & IDE that attributes like `th:text`, `th:if`, `th:each` belong to Thymeleaf.
- Without this, Thymeleaf-specific attributes will not work.

2. `th:text`

- Replaces the content of an HTML tag with a **value from the Model**.
- Syntax: `<tag th:text="${variable}"></tag>`
- `${variable}` → Fetches the value of the variable passed from the Controller.
- If a variable is not found, it shows the default text written inside the tag.

📒 Arithmetic Operation

You can directly perform mathematical calculations inside `${...}`.

Examples:

```
<p th:text="${10 + 5}"></p>         <!-- 15 -->
<p th:text="${10 - 3}"></p>         <!-- 7 -->
<p th:text="${10 * 2}"></p>         <!-- 20 -->
<p th:text="${20 / 4}"></p>         <!-- 5 -->
<p th:text="${20 % 3}"></p>         <!-- 2 (modulus) -->
```

Works with variables too:

```
<p th:text="${a + b}"></p>   <!-- if a=5, b=10 → 15 -->
<p th:text="${price * quantity}"></p>
```

📒 Variables In Thymeleaf

Using Model Variables

Variables added in the Controller with `model.addAttribute()` can be accessed in templates.

Example (Controller):

```
model.addAttribute("num1", 50);
model.addAttribute("num2", 25);
```

Example (HTML):

```
<p th:text="${num1 + num2}"></p>   <!-- 75 -->
<p th:text="${num1 - num2}"></p>   <!-- 25 -->
```

🔹 Variable Default / Safe Navigation

- `${var}` → If the variable exists, print it.
- `${var ?: 'Default Value'}` → If the variable is null, print default.
- `${var?.property}` → Safe navigation (avoid null pointer).
  - Var → is java object
  - property → property is the java objets property.

`th:with`

Definition

- `th:with` is used to define local variables inside a tag.
- These variables are valid only within that tag and its children.
- It helps avoid repeating expressions and makes code cleaner.

Example

```
<div th:with="x= ${num1}, y= ${num2}">
    <p th:text="${x}" ></p>
    <p th:text="${y}" ></p>
    <p th:text="${x + y}" ></p>
</div>
```

`x` and `y` exist only inside this `<div>`.
Outside, they won't be accessible.

📒 Utility Objects

Definition

- Thymeleaf provides built-in **utility objects** (prefixed with `#`) that give extra functions for common tasks.
- They can be used inside expressions `${...}`.
- Examples: `#strings`, `#numbers`, `#dates`, `#lists`, etc.

Commonly Used Utility Objects

```
@Controller
public class UtilityObjectController {
        @GetMapping("/utilityObjects")
        public String utilityObjects(Model m) {

                m.addAttribute("name", "Gaurav");

                m.addAttribute("number", 10);
                m.addAttribute("nPercent", 0.85);

                m.addAttribute("today", new Date());

                List<Integer> listNumbers = List.of(1 , 2, 3);
                m.addAttribute("listNumbers", listNumbers);

                return "utilityObjects";
        }
}
```

1) #strings → String Operations

```
<p th:text="${#strings.toUpperCase(name)}"></p>
<p th:text="${#strings.length(name)}"></p>
```

2) `#numbers` → Number Operations

```
<p th:text="${#numbers.formatInteger(number, 3)}"></p>
<p th:text="${#numbers.formatPercent(nPercent, 1, 2)}"></p>
```

3) `#dates` → Date & Time Operations

```
<p th:text="${#dates.format(today, 'dd-MM-yyyy')}"></p>
```

4) `#lists` → List Operations

```
<p th:text="${#lists.size(listNumbers)}"></p>
<p th:text="${#lists.isEmpty(listNumbers)}"></p>
```

Note : These are just some examples of utility object methods.
Thymeleaf provides many more methods in each utility object

📒 Iteration In Thymeleaf

- Iteration means **looping over a collection** (list, array, set, map) in Thymeleaf.
- Done using the attribute:

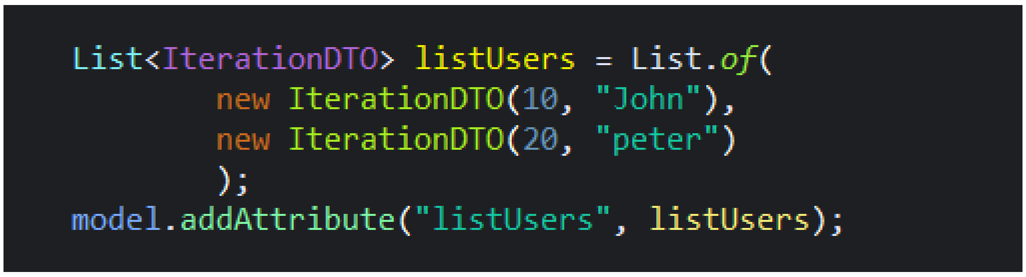
```
th:each="variable : ${collection}"
```

1. Basic Example :

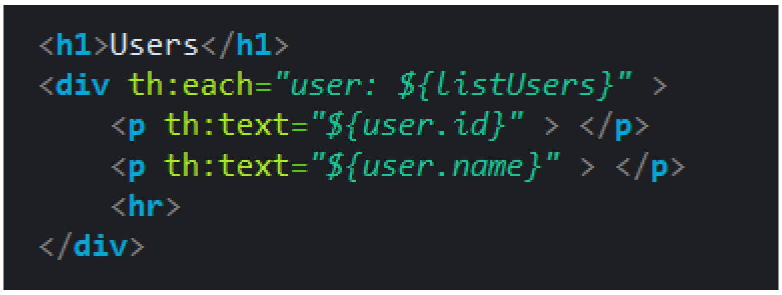
```
List<String> fruits = List.of("Apple", "Mango", "Banana");
model.addAttribute("fruits", fruits);
```

```
<h1>Fruits</h1>
<ul>
    <li th:each="fruit: ${fruits}" th:text="${fruit}" />
</ul>
<hr>
```

2. Iteration with Object Properties :



```java
List<IterationDTO> listUsers = List.of(
        new IterationDTO(10, "John"),
        new IterationDTO(20, "peter")
        );
model.addAttribute("listUsers", listUsers);
```



```html
<h1>Users</h1>
<div th:each="user: ${listUsers}" >
    <p th:text="${user.id}" > </p>
    <p th:text="${user.name}" > </p>
    <hr>
</div>
```

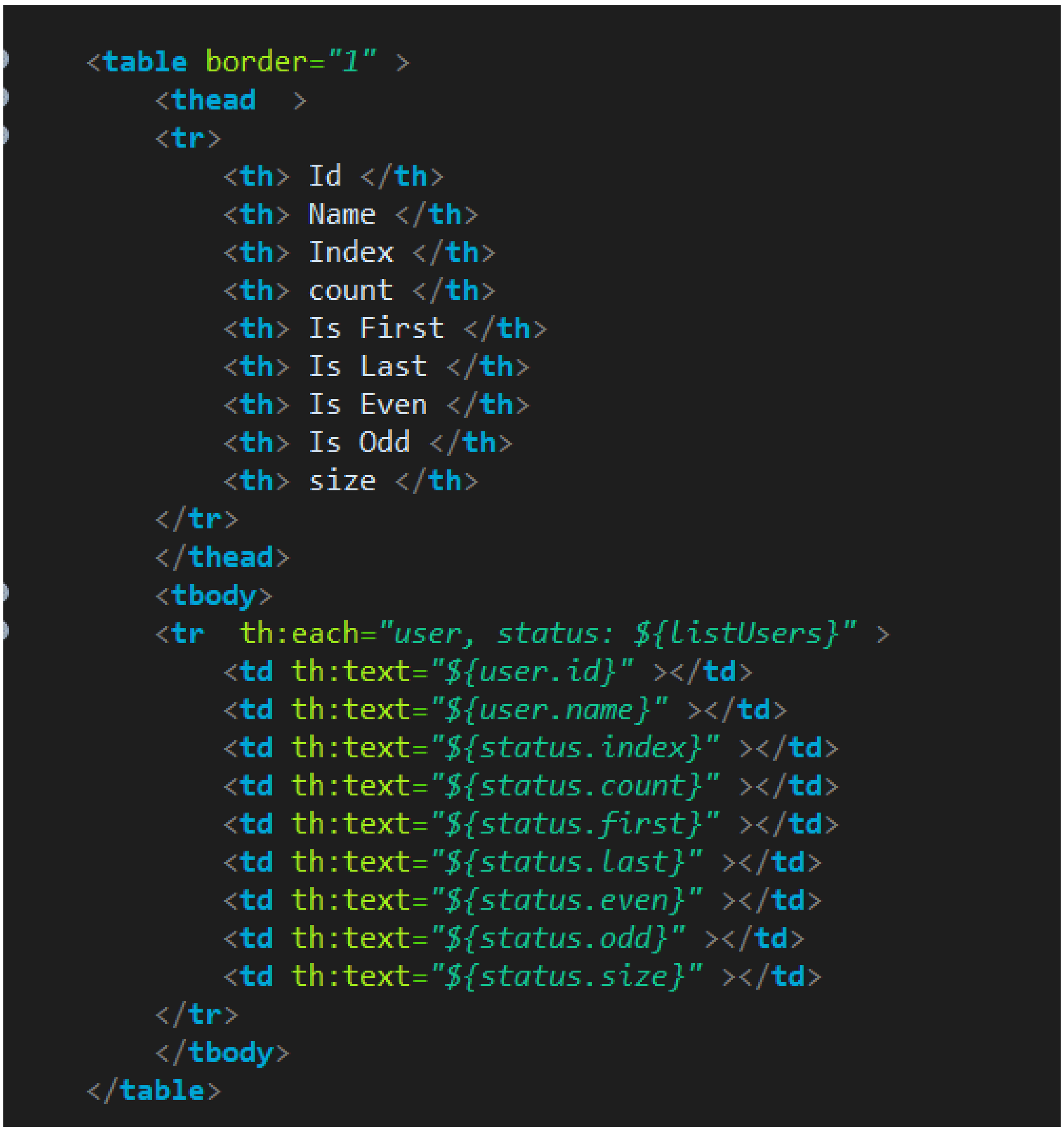
3. Iteration Status

Thymeleaf provides a loop status variable with useful info (index, count, first, last, even, odd).

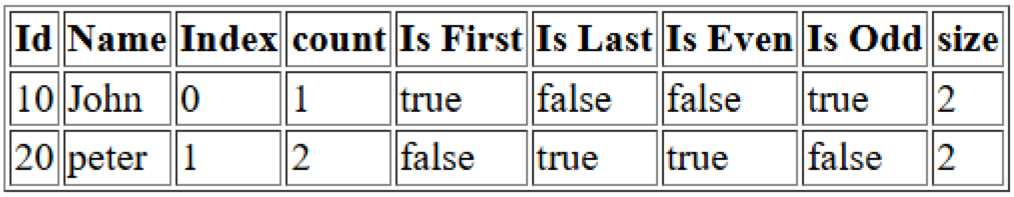
syntax : th:each="item, status : ${items}"

`status.index` → 0-based index
`status.count` → 1-based index
`status.size` → total elements
`status.first` → true if first element
`status.last` → true if last element
`status.even` → true if index is even
`status.odd` → true if index is odd

you can give any name to the status variable.



```
    <table border="1" >
        <thead  >
        <tr>
            <th> Id </th>
            <th> Name </th>
            <th> Index </th>
            <th> count </th>
            <th> Is First </th>
            <th> Is Last </th>
            <th> Is Even </th>
            <th> Is Odd </th>
            <th> size </th>
        </tr>
        </thead>
        <tbody>
        <tr  th:each="user, status: ${listUsers}" >
            <td th:text="${user.id}" ></td>
            <td th:text="${user.name}" ></td>
            <td th:text="${status.index}" ></td>
            <td th:text="${status.count}" ></td>
            <td th:text="${status.first}" ></td>
            <td th:text="${status.last}" ></td>
            <td th:text="${status.even}" ></td>
            <td th:text="${status.odd}" ></td>
            <td th:text="${status.size}" ></td>
        </tr>
        </tbody>
    </table>
```



Id	Name	Index	count	Is First	Is Last	Is Even	Is Odd	size
10	John	0	1	true	false	false	true	2
20	peter	1	2	false	true	true	false	2

📒 Conditional Statements In Thymeleaf

Thymeleaf provides attributes to handle **if-else conditions** directly in the template. These are useful when you want to show/hide elements or display alternate content based on some conditions.

- `th:if` → show only if condition true
- `th:unless` → show only if condition false
- `th:if + th:unless` → works like if-else
- `?:` → ternary operator
- `th:switch / th:case` → switch-case handling

1. `th:if`

- Displays the element **only if** the condition is `true`.
- If the condition is `false`, the element is completely removed from the HTML.

2. `th:unless`

- Opposite of `th:if`.
- Displays the element **only if the condition is `false`**.

```
<h2 th:if="${age} >= 18" >You are an adult</h2>

<h2 th:unless="${age} >= 18" >You are not an adult</h2>
```

3. Conditional Expressions (Ternary Operator)

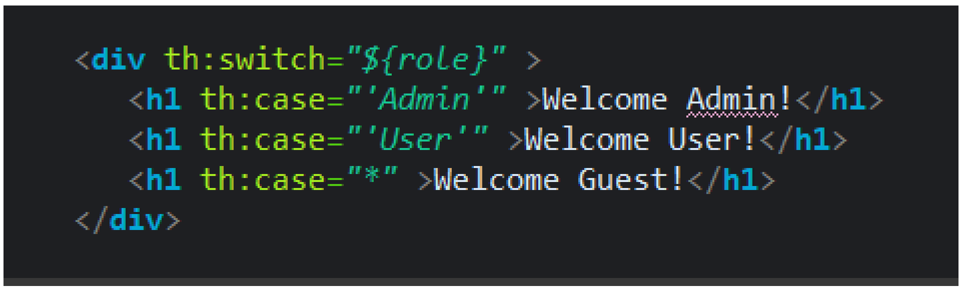
You can use the ternary operator `condition ? valueIfTrue : valueIfFalse`.

```
<h1 th:text="${isActive} ? 'Active' : 'Inactive' "
    th:style="${isActive} ? 'color: green;' : 'color: red;' "
 ></h1>
```

👉 If `isActive = true`, then "Active" will be displayed. Otherwise, "Inactive".

5. `th:switch` (Switch Case with)

- Works like Java’s `switch`.
- Used with `th:case`.



```html
<div th:switch="${role}" >
   <h1 th:case="'Admin'" >Welcome Admin!</h1>
   <h1 th:case="'User'" >Welcome User!</h1>
   <h1 th:case="*" >Welcome Guest!</h1>
</div>
```

👉 If `user.role = ADMIN`, it will show "Welcome Admin!".
👉 If none matches, `*` (default case) is shown.

📒 Thymeleaf Fragment and Insertion

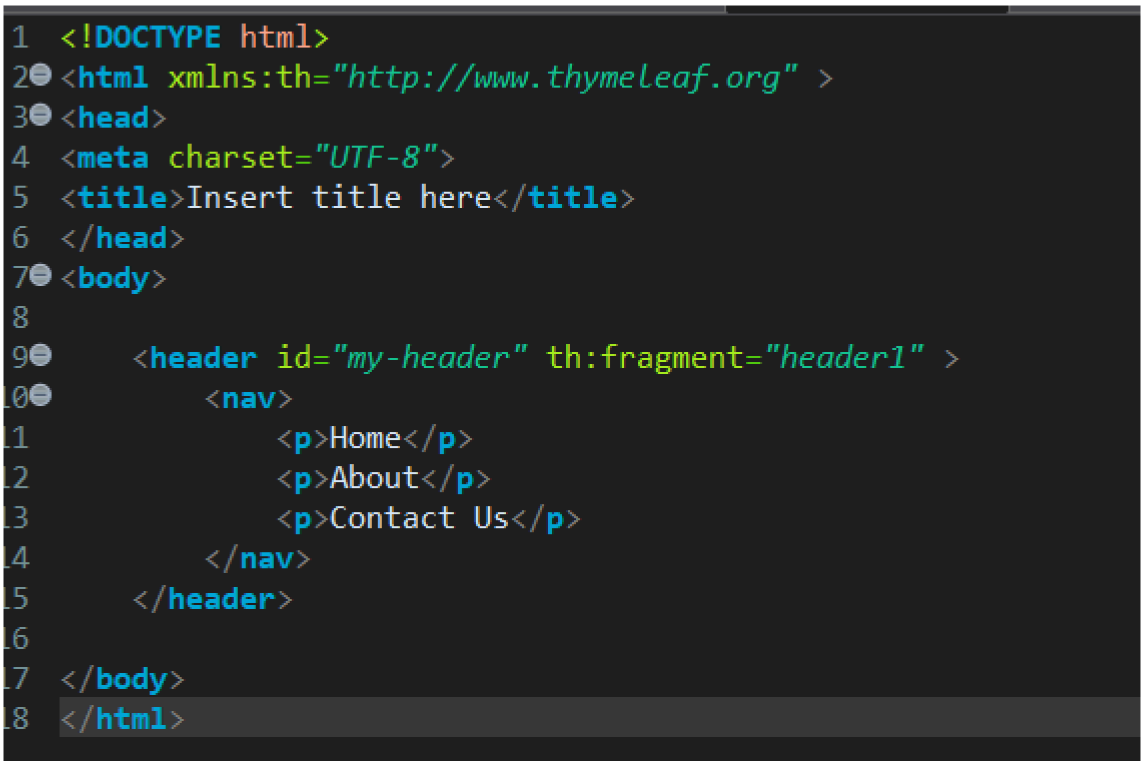
What is a Fragment?

- A fragment in Thymeleaf is a reusable piece of HTML code.
- You can define it once (like a header, footer, or navigation bar) and reuse it across multiple pages.
- This avoids code duplication and makes templates easier to maintain.
- Declared using `th:fragment` and included with `th:insert`, `th:replace`, or `th:include`.

Syntax (Defining a fragment):

```
<html-element th:fragment="unique-fragment-name">

   <!-- reusable content -->

</html-element
```

—> header.html



```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org" >
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>

    <header id="my-header" th:fragment="header1" >
        <nav>
            <p>Home</p>
            <p>About</p>
            <p>Contact Us</p>
        </nav>
    </header>

</body>
</html>
```

Using a fragment in another page

- `th:insert`
- `th:replace`
- `th:include`.

Syntax :

<html-element th:insert/replace/include =”fragment-html-name :: fragment-name” />

If our fragment html name is header.html and fragment name is my-header, then tag will be

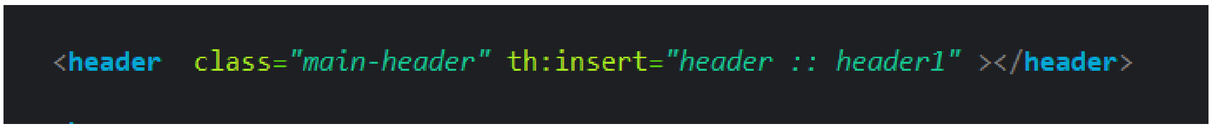
<html-element th:insert/replace/include =”header :: my-header” />

⚡ Note: No need to add file extension.

a) th:insert

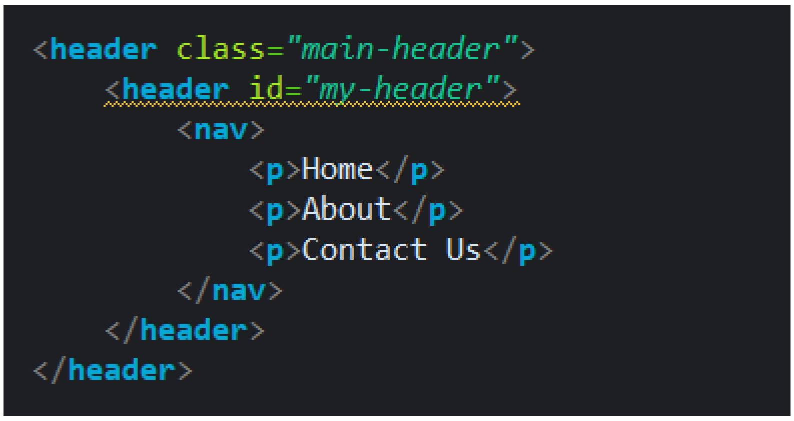
- Inserts the fragment inside the host tag (keeps the surrounding tag).
- Good when you want to keep your host tag.

—> main html



```
<header  class="main-header" th:insert="header :: header1" ></header>
```

Result



```
<header class="main-header">
    <header id="my-header">
        <nav>
            <p>Home</p>
            <p>About</p>
            <p>Contact Us</p>
        </nav>
    </header>
</header>
```

I will keep the tag from the main html page and include the fragment tag inside of it.

b) th:replace

- Replaces the **host tag completely** with the fragment.
- Good when the fragment itself is the whole element.

—> main html

```
<header class="main-header" th:replace="header :: header1" ></header>
```

Result :

```
<header id="my-header">
    <nav>
        <p>Home</p>
        <p>About</p>
        <p>Contact Us</p>
    </nav>
</header>
```

It will replace the tag from main jsp with tag from the fragment jsp

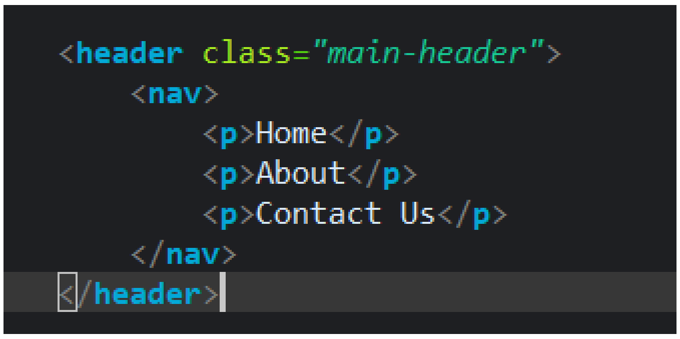
c) th:include (older, not recommended much)

- Keeps the host tag and only includes the content from the fragment tag and not the actual fragment tag

—> main html

```
<header class="main-header" th:include="header :: header1" ></header>
```

Result :

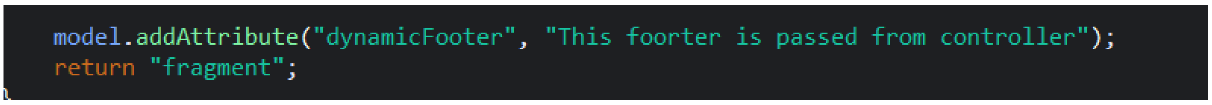


```html
<header class="main-header">
    <nav>
        <p>Home</p>
        <p>About</p>
        <p>Contact Us</p>
    </nav>
</header>
```

It kept the host tag and only added content from fragment tag

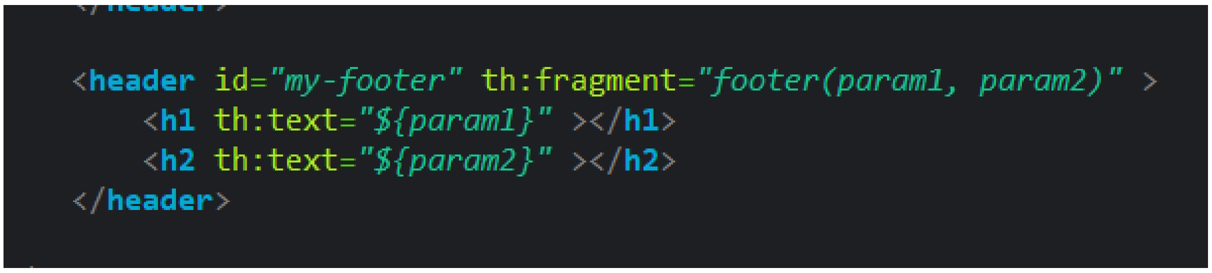
Passing Dynamic Values to Fragment

Thymeleaf allows you to pass **dynamic values** (variables, expressions, method results, etc.) from your model to the HTML.



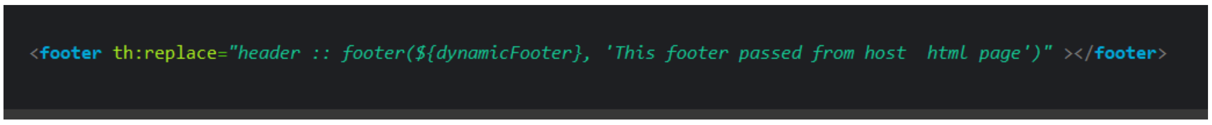
```java
model.addAttribute("dynamicFooter", "This foorter is passed from controller");
return "fragment";
```

——--> Fragment html



```html
<header id="my-footer" th:fragment="footer(param1, param2)" >
    <h1 th:text="${param1}" ></h1>
    <h2 th:text="${param2}" ></h2>
</header>
```

——--------->



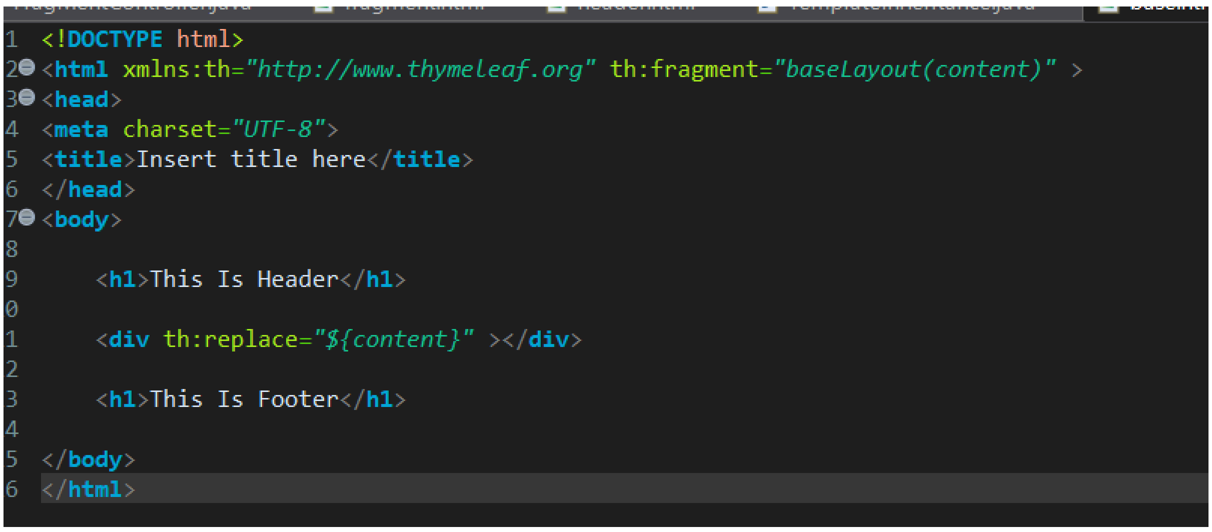
```html
<footer th:replace="header :: footer(${dynamicFooter}, 'This footer passed from host  html page')" ></footer>
```

📒 Thymeleaf Template Inheritance

What is Template Inheritance?

- In Thymeleaf, template inheritance allows you to define a **base layout** (master template) and reuse it across multiple pages.
- Child templates can **extend** the base and override or add specific sections.
- Helps in **code reusability** and **consistent UI**.

Base Template (e.g., `base.html`)



```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org" th:fragment="baseLayout(content)" >
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>

    <h1>This Is Header</h1>

    <div th:replace="${content}" ></div>

    <h1>This Is Footer</h1>

</body>
</html>
```

`th:fragment="baseLayout(content)"`

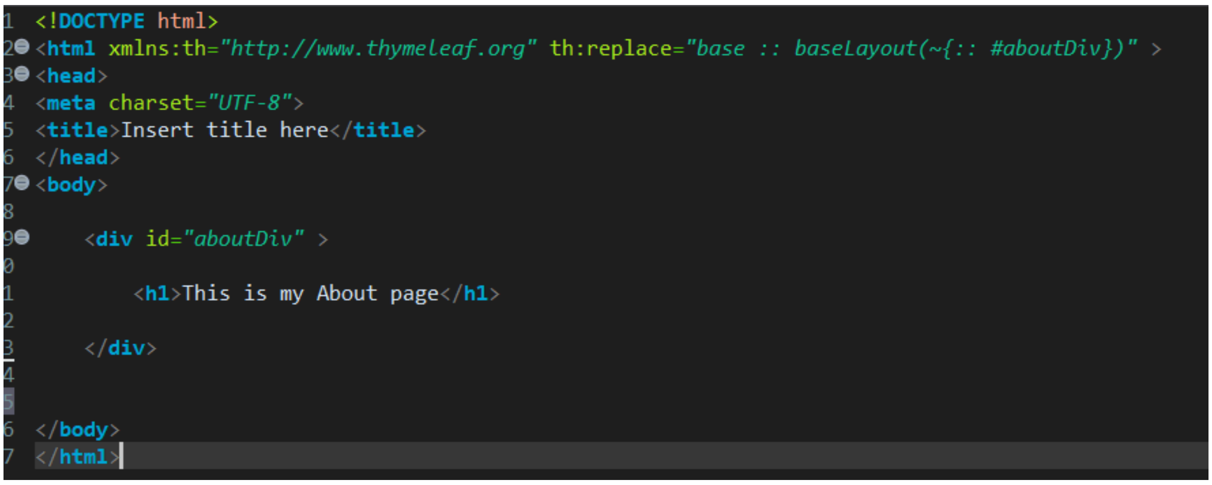
- Defines a **fragment** (reusable template block).
- Here, the fragment's name is `baseLayout`.
- It accepts a **parameter** called `content` → this parameter will be replaced dynamically with page-specific content.

`<h1>This Is Header</h1>` & `<h1>This Is Footer</h1>`

- Static parts of the layout → always shown on every page.
- This creates a **common structure (header + footer)**.

`<div th:replace="${content}"></div>`

- Acts as a **placeholder**.
- Whatever is passed as `content` from child pages will be inserted here.



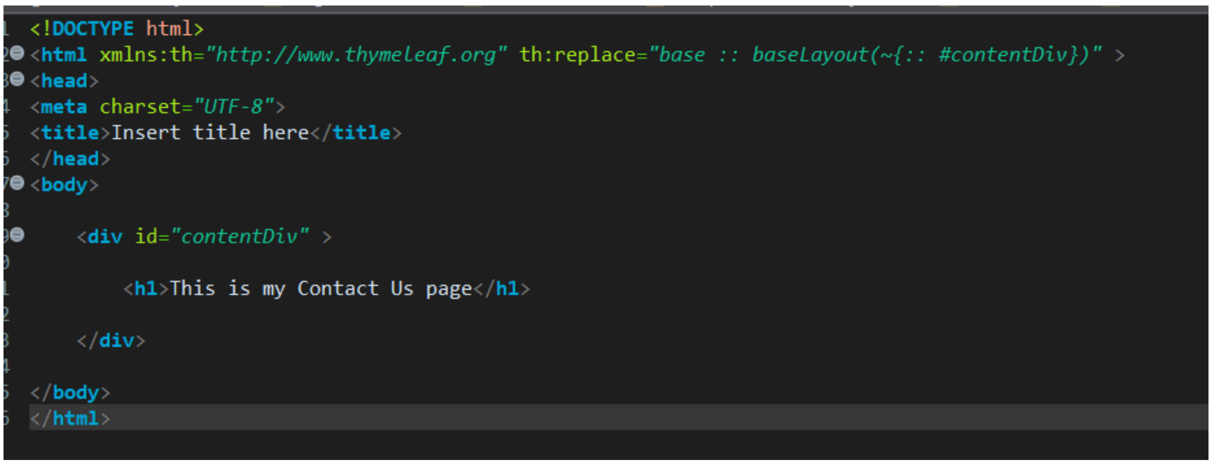
```html
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org" th:replace="base :: baseLayout(~{:: #aboutDiv})" >
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>

    <div id="aboutDiv" >

        <h1>This is my About page</h1>

    </div>


</body>
</html>
```

`th:replace="base :: baseLayout(~{:: #aboutDiv})"`

- This tells Thymeleaf:
  → Go to `base.html`.
  → Use the fragment `baseLayout`.
  → Replace the `content` parameter with the current page's `#aboutDiv`.

`~{:: #aboutDiv}`

- Means → take the `div` with `id="aboutDiv"` from this page and inject it into the base layout's `content` placeholder.



```html
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org" th:replace="base :: baseLayout(~{:: #contentDiv})" >
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>

    <div id="contentDiv" >

        <h1>This is my Contact Us page</h1>

    </div>

</body>
</html>
```

📒 @{...}

In **Thymeleaf**, the `@{...}` syntax is used for **URL expressions**. It helps you generate context-relative, dynamic, and properly encoded URLs in your HTML templates.

1. **Basic usage**
    - <a th:href="@{/home}">Home</a>
    - If your app is deployed at `http://localhost:8080/myapp`, this will render as:
    - <a href="/myapp/home">Home</a>

2. **With query parameters**
    - <a th:href="@{/search(q=${keyword})}">Search</a>
    - If `keyword = "spring"`, then:
    - <a href="/myapp/search?q=spring">Search</a>

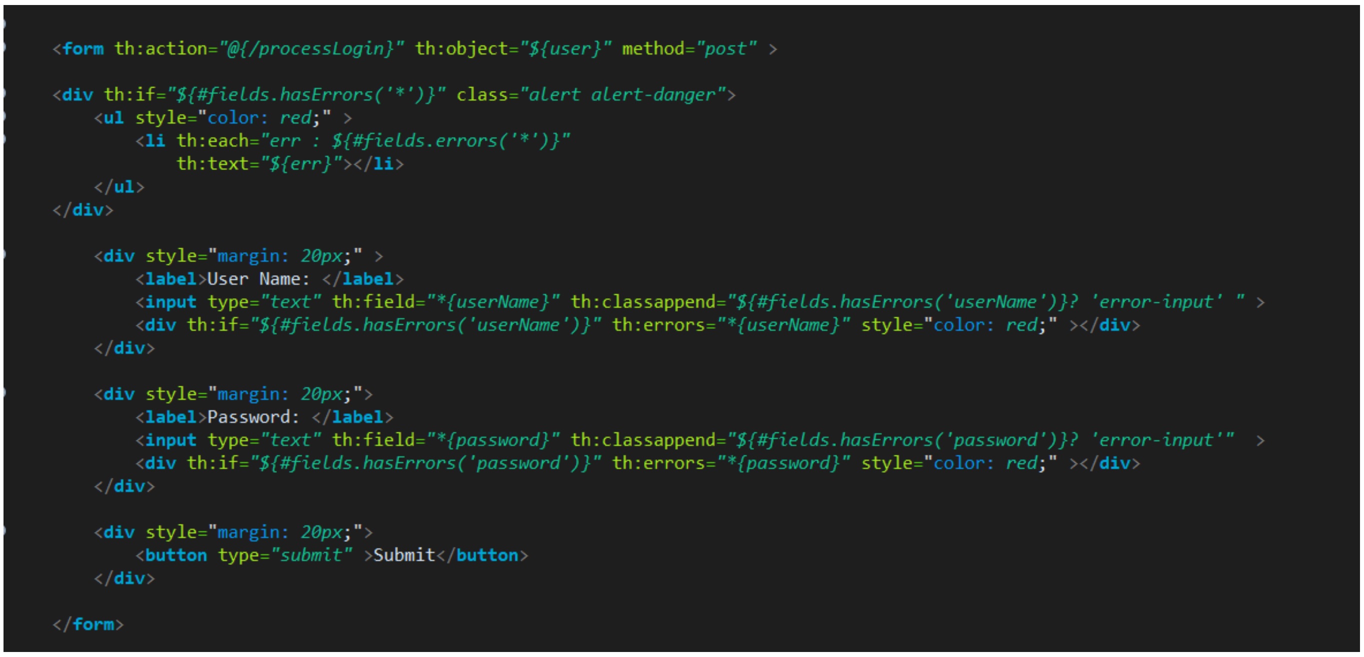
3. **With multiple query parameters**
    - <a th:href="@{/filter(cat=${category}, sort=${sortType})}">Filter</a>
    - If `category = "books"` and `sortType = "price"`, result:
    - <a href="/myapp/filter?cat=books&sort=price">Filter</a>

4. **With path variables**
    - Syntax: `@{/path/{var}(var=${value})}`
    - <a th:href="@{/user/{id}(id=${user.id})}">Profile</a>
    - If `user.id = 101`:
    - <a href="/myapp/user/101">Profile</a>

6. **Absolute URLs**
    - <a th:href="@{http://example.com/about}">About</a>
    - <a href="http://example.com/about">About</a>

📒 Thymeleaf With Html Form

Thymeleaf provides the `th:object` and `th:field` attributes to bind form data with backend model objects.



```html
<form th:action="@{/processLogin}" th:object="${user}" method="post" >

<div th:if="${#fields.hasErrors('*')}" class="alert alert-danger">
    <ul style="color: red;" >
        <li th:each="err : ${#fields.errors('*')}"
            th:text="${err}"></li>
    </ul>
</div>

    <div style="margin: 20px;" >
        <label>User Name: </label>
        <input type="text" th:field="*{userName}" th:classappend="${#fields.hasErrors('userName')}? 'error-input' " >
        <div th:if="${#fields.hasErrors('userName')}" th:errors="*{userName}" style="color: red;" ></div>
    </div>

    <div style="margin: 20px;">
        <label>Password: </label>
        <input type="text" th:field="*{password}" th:classappend="${#fields.hasErrors('password')}? 'error-input'"  >
        <div th:if="${#fields.hasErrors('password')}" th:errors="*{password}" style="color: red;" ></div>
    </div>

    <div style="margin: 20px;">
        <button type="submit" >Submit</button>
    </div>

</form>
```

- `th:action="@{/register}"` → maps form submission to `/register` endpoint.
- `th:object="${user}"` → binds form with model attribute `user`.
- `th:field="*{username}"` → automatically binds to `user.getUsername()` and `setUsername()`.
- `#fields.hasErrors('fieldName')` → checks if a field has errors.
- `th:errors="*{field}"` → displays validation error message.
- `th:classappend` adds one or more CSS classes to an element's existing `class` attribute dynamically based on a condition or expression.

Model Class with Validation

```java
public class UserDTO {
@NotBlank(message = "User Name can not be empty")
@Size(min =  3, message = "User Name should have min 3 characters")
private String userName;

@Size(min = 3, max = 8, message = "password shold have mininum 3 and maximum 8
characters")
private String password;
```

Controller

```
        @Controller
        public class FormValidationController {
        @GetMapping("/showLogin")
        public String showLogin(Model model) {

                UserDTO userDTO = new UserDTO();
                userDTO.setUserName("Gaurav");
                userDTO.setPassword("12345678");
                model.addAttribute("user", userDTO);
                return "login";
        }

        @PostMapping("/processLogin")
        public String processLogin(@Valid @ModelAttribute("user") UserDTO user, BindingResult result) {

                if (result.hasErrors()) {
                        return "login";
                }
                return "wlcome";
        }
}
```

Displaying All Errors Together

```
<div th:if="${#fields.hasErrors('*')}" >
    <ul style="color: red;" >
      <li th:each="err : ${#fields.errors('*')}"
        th:text="${err}"></li>
    </ul>
  </div>
```

📒 Thymeleaf Static Resources (CSS, JS, Images)

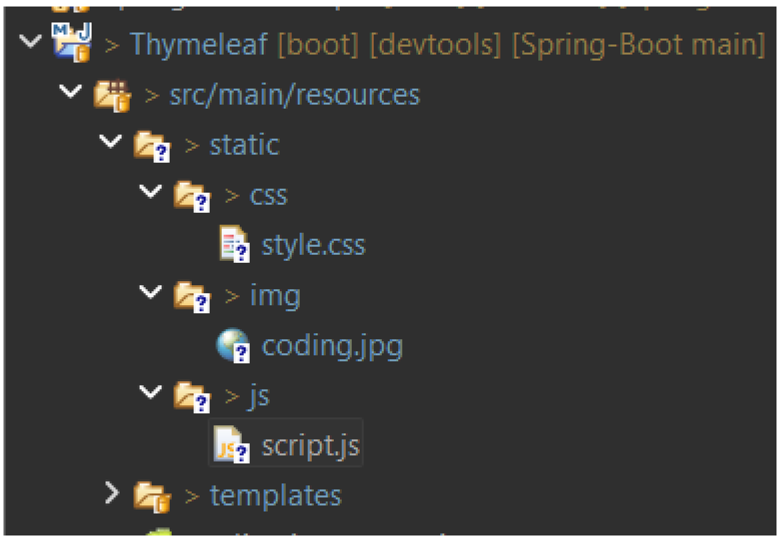
Static resources are files like **CSS, JavaScript, and Images** that are stored in the project and served to the client.
Thymeleaf provides the `@{...}` syntax to properly link these resources with the correct context path.

1. **Where to put static resources?**

```
src/main/resources/static/
```

Inside this folder, you can create subfolders:

- `/css/` → for stylesheets
- `/js/` → for JavaScript files
- `/images/` → for images



```
> Thymeleaf [boot] [devtools] [Spring-Boot main]
  > src/main/resources
    > static
      > css
          style.css
      > img
          coding.jpg
      > js
          script.js
    > templates
```

2. **Including CSS**

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

- Use th:href instead of href

3. **Including JavaScript**

```
<script type="text/javascript" th:src="@{/js/script.js}" ></script>
```

- Use th:src instead of src

4. Including Images

```
<img alt="image" th:src="@{/img/coding.jpg}" >
```

- Use th:src instead of src

6. Absolute URLs

```
<script th:src="@{https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/js/bootstrap.bundle.min.js}"></script>
```