Basic Thymeleaf Tags

1. Basic Operation Tags

1.1. Text Replacement

The most basic operation allows us to display a value from the server inside an HTML tag. To do this, we add the `th:text` attribute and wrap the server variable in `${...}`. If the passed text is a date (LocalDate, LocalDateTime), we can format it using `#temporals.format(,)`:

```
<p th:text="${#temporals.format(fecha, 'dd-MM-yyyy HH:mm')}">2024-12-31</p>
```

Remember that if we use the `"` character for the template instruction, we must use `'` for strings inside it. This is common in most server-side languages (like PHP).

1.2. Conditions

We can use conditions to control whether a part of our HTML is displayed. For this, we use the `th:if` and `th:unless` attributes. `th:unless` works like an `else`—technically it means “if not”—and can be used independently. It supports the same comparison operators as Java (`>` `>=` `<` `<=` `==` `!=`).

```
<div th:if="${result>10}">Result greater than 10</div>
<div th:unless="${result>10}">Result less than or equal to 10</div>
```

Or:

```
<div th:if="${result>10}">
    <span th:text="${result}">*</span> is greater than 10
</div>
<div th:unless="${result>10}">
    <span th:text="${result}">*</span> is less than or equal to 10
</div>
```

> **NOTE:** The `<span>` tag is an inline element, meaning it flows inside existing text without creating a line break.

> A crucial distinction in HTML is between inline elements and block-level elements:
>
> - **Inline elements:** Elements like `<span>` , `<a>` , and `<strong>` reside within the natural flow of a paragraph or sentence. They only occupy the space required for their content.
> - **Block-level elements:** Elements like `<div>` , `<h1>` , and `<p>` start on a new line and occupy the full width of their parent container.
>
> ***source***

Another way to implement a comparison is using the `${}? '':''` operator, which works like a ternary operator (`${condition}?'True':'False'`).

```
<span th:text="${result>10} ? 'Result greater than 10' : 'Result less than or equal to
```

Logical operators in Thymeleaf are **not the same as in Java**; they are written as words: `and` , `or` , `not` .

```
<p th:if="${age > 18 and registered == true}">Login successful</p>
<p th:if="${age < 18 or registered != true}">Invalid user</p>
```

1.3. Iterations

To iterate over a collection, we use `th:each` , which works like the `for(:)` operator in Java:

```
<p th:each="product : ${productList}">
    <span th:text="${product}">default</span>
</p>
```

In this example, `productList` can be any collection of basic types (like String). If it contains objects with attributes, we can access them as long as they have standard getters (`getId()` , `getName()` , etc.):

```
<table>
<tr th:each="employee : ${employeeList}">
    <td th:text="${employee.id}">id</td>
    <td th:text="${employee.name}">name</td>
</tr>
</table>
```

We can also add a counter variable using an iterator with two attributes: `index` (starts at 0) and `count` (starts at 1):

```
<table>
<tr th:each="employee, iter : ${employeeList}">
    <td th:text="${iter.count}">counter</td>
    <td th:text="${employee.id}">id</td>
    <td th:text="${employee.name}">name</td>
</tr>
</table>
```

2. Fragments and Blocks

Fragments are blocks of code that can be saved in a file and reused across different pages to avoid code duplication. Typical uses include `<head>` , footers, or general menus that appear on multiple pages.

Each fragment is identified with the attribute `th:fragment="fragmentName"` . You can include multiple fragments in one file. Header-related code goes in `<head>` , and body-related code goes in `<body>` .

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head th:fragment="header">
    <meta charset="UTF-8" />
    <title>Title for all pages</title>
    <link th:href="@{/css/styles.css}" rel="stylesheet">
</head>
<body>
    <footer th:fragment="footer">
        <a href="https://www.mycompany.com">&copy; My Company</a>
    </footer>
</body>
</html>
```

By default, these files go in the `/templates` folder. If you create many fragment files, it's advisable to make a subfolder.

Notice the `<link>` uses `th:href="@{/css/styles.css}"` . This tells Thymeleaf to convert `@{...}` into a URL relative to the application context (e.g., `/myapp/css/styles.css`).

Using Fragments in Other Pages

Once fragments are created, you can reuse them with `th:replace` to **replace the tag** or `th:insert` to **insert the content without replacing the tag**. Unlike variable expressions `${...}`, fragment expressions use `~{...}`. On a Spanish keyboard, `~` is typed with `Alt Gr + Ñ`.

Example using `replace`:

```
<head th:replace="~{/fragments.html::header}"></head>
```

Or using `insert`:

```
<head th:insert="~{/fragments.html::header}"></head>
```

- Using `insert` in `<head>` will duplicate the `<head>` tag.
- To avoid this, use **blocks** with `<th:block>`:

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <th:block th:fragment="header">
        <meta charset="UTF-8" />
        <title>Title for all pages</title>
        <link th:href="@{/css/styles.css}" rel="stylesheet">
    </th:block>
</head>
</html>
```

When inserting:

```
<head>
    <th:block th:insert="~{/fragments.html::header}"></th:block>
</head>
```

- `<th:block>` does **not** appear in the final HTML; it only groups fragment content.
- **`<th:block>` is the only Thymeleaf-specific tag**; the rest are standard HTML elements with `th:*` attributes.

Fragments can also include other Thymeleaf tags:

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <th:block th:fragment="header">
        <meta charset="UTF-8" />
        <title th:text="${title}">*</title>
        <link th:href="@{/css/styles.css}" rel="stylesheet">
    </th:block>
</head>
</html>
```

3. Null Handling

If a server-provided object is `null`, accessing its attributes would cause an exception. Thymeleaf provides mechanisms to avoid this.

Safe Navigation Operator `?.`

Displays an empty tag if the object is `null`:

```
<td th:text="${employee?.name}">name</td>
```

Elvis Operator `?:`

Displays a default value if the object is `null`:

```
<td th:text="${employee.name} ?: 'no name'">*</td>
```

> This is known as the *Elvis* operator.

> **NOTE:** The `*` is optional and just serves as a placeholder in the static HTML.

4. Thymeleaf and CSS

4.1. `th:style` and `th:classappend`

Thymeleaf allows dynamic CSS attributes and classes:

```
<table th:style="${number>0} ? 'display:block' : 'display:none'">
    (...)
</table>
```

```
<span th:classappend="${number>50} ? 'highNumber' : 'lowNumber'" th:text="${number}">*<
```

- Adds `display:block` if number > 0.
- Adds `highNumber` class if number > 50, else `lowNumber`.

4.2. Difference between `th:src` / `th:href` and `src` / `href`

- `th:href` and `th:src` convert `@{...}` into URLs relative to the **application context**.
- `href` and `src` use the literal path.

Rules of thumb:

- Project-relative resources (CSS, JS, images, Spring controllers) → use `th:href`, `th:src`.
- Absolute external URLs (CDNs, other websites) → use `href`, `src`.

Spring Boot + Thymeleaf Example

- File location: `src/main/resources/static/css/styles.css`
- Static files are served directly from `/static`.

In your template:

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

- Thymeleaf generates the URL relative to the context path:
  - If app runs at `/` → `/css/styles.css`
  - If app runs at `/myapp` → `/myapp/css/styles.css`
- Spring Boot serves the file from `static/css/styles.css`.

Using plain `href="/css/styles.css"` may fail if the context path is not `/`. `th:href` adjusts automatically.

> **ACTIVITY:**
> Create a new project based on Frédéric Chopin. Pass dynamic data to Thymeleaf templates using a `Model`.
>
> - The homepage can display the current year (e.g., @2025) using `LocalDate.now()`.
> - On the repertoire page, list pieces using a class or record `MusicalPiece` with title, composer, year, instrumentation.
> - Later, this data will be stored in a database.
>
> Adapt all views where possible.
>
> Your exercise must meet the following requeriments:
>
> - **Create a data model to save the data about the Musical Pieces**. You can use classes, records, arrays or anything similar. Keep in mind you need to use a Collection based on this type of data (so, classes or records will be easier).
> - **Use fragments** for the `<head>`. Use at least once the tag `<th:block>`.
> - **Use the Elvis operator** to check if a value is null or not.
> - Use at least once a condition and an iteration.
>   - For instance, as condition you may use a different css class is a musical piece is a *solo piano piece* or a *piano concert*.
>   - Of course, the easiest place to implement the iteration is listing the musical pieces.
> - Set `spring.thymeleaf.cache=false` in `application.properties` and the `xmlns` attribute in the `<html>` tag.
> - **Use `th:href` and `th:src`** instead of HTML `href` and `src`.
>
> You have to explain how you have done all of this in the memory you will deliver along the project.

▶ **Previous Activities**