Thymeleaf CheatSheet

Created	@Oct 6, 2020 5:09 PM
Tags	
Updated	@Oct 6, 2020 5:15 PM

Thymeleaf is a modern server-side Java template engine for both web and standalone environments.

Thymeleaf's main goal is to bring elegant *natural templates* to your development workflow — HTML that can be correctly displayed in browsers and also work as static prototypes, allowing for stronger collaboration in development teams.

Most Thymeleaf attributes allow their values to be set as or containing *expressions*, which we will call *Standard Expressions* because of the dialects they are used in. These can be of five types:

- `${...}` : Variable expressions.
- `*{...}` : Selection expressions.
- `#{...}` : Message (i18n) expressions.
- `@{...}` : Link (URL) expressions.
- `~{...}` : Fragment expressions.

Variable expressions

Variable expressions are OGNL expressions –or Spring EL if you're integrating Thymeleaf with Spring– executed on the *context variables* — also called *model attributes* in Spring jargon. They look like this:

```
${session.user.name}
```

And you will find them as attribute values or as a part of them, depending on the attribute:

```
<span th:text="${book.author.name}">
```

Selection expressions

Selection expressions are just like variable expressions, except they will be executed on a previously selected object instead of the whole context variables map. They look like this:

```
*{customer.name}
```

The object they act on is specified by a `th:object` attribute:

```
<div th:object="${book}">
  ...
  <span th:text="*{title}">...</span>
  ...
</div>
```

So that would be equivalent to:

```
{
  // th:object="${book}"
  final Book selection = (Book) context.getVariable("book");
  // th:text="*{title}"
  output(selection.getTitle());
}
```

Message (i18n) expressions

Message expressions (often called *text externalization*, *internationalization* or *i18n*) allows us to retrieve locale-specific messages from external sources (`.properties` files), referencing them by a key and (optionally) applying a set of parameters.

In Spring applications, this will automatically integrate with Spring's `MessageSource` mechanism.

```
#{main.title}
```

```
#{message.entrycreated(${entryId})}
```

You can find them in templates like:

```
<table>
  ...
  <th th:text="#{header.address.city}">...</th><th th:text="#{header.address.country}">...</th>
  ...
</table>
```

Link (URL) expressions

Link expressions are meant to build URLs and add useful context and session info to them (a process usually called *URL rewriting*).

So for a web application deployed at the `/myapp` context of your web server, an expression such as:

```
<a th:href="@{/order/list}">...</a>
```

Could be converted into something like this:

```
<a href="/myapp/order/list">...</a>
```

Or even this, if we need to keep sessions and cookies are not enabled (or the server doesn't know yet):

```
<a href="/myapp/order/list;jsessionid=23fa31abd41ea093">...</a>
```

URLs can also take parameters:

```
<a th:href="@{/order/details(id=${orderId},type=${orderType})}">...</a>
```

Resulting in something like this:

```
<!-- Note ampersands (&) should be HTML-escaped in tag attributes... -->
<a href="/myapp/order/details?id=23&amp;type=online">...</a>
```

Fragment expressions

Fragment expressions are an easy way to represent fragments of markup and move them around templates. Thanks to these expressions, fragments can be replicated, passed to other templates as arguments, and so on.

The most common use is for fragment insertion using `th:insert` or `th:replace`:

```
<div th:insert="~{commons :: main}">...</div>
```

But they can be used anywhere, just as any other variable:

```
<div th:with="frag=~{footer :: #main/text()}"><p th:insert="${frag}"></div>
```

Thymeleaf cheat sheet

This is a cheat sheet to summarize all the main Thymeleaf features and how to use them to kickstart you with Thymeleaf.

What is Thymeleaf

Thymeleaf is an engine that builds dynamic pages from templates that are written in XHTML with the help of some special attributes, so it is a **template engine**.

A template engine in Java is an engine that parses XHTML pages which contain special tags and attributes or syntax. Those attributes will then use some variables to build the web page, and are bound to fields of Java beans on the server side. The engine will resolve those variables in the attributes to their actual values, then process the page according to those values and build a normal HTML page.

For example, the templating engine will resolve the values of the list of students which was passed through the controller, and then use those values to replace the attributes and dummy text with actual data from the fetched list.

Thymeleaf is an **in-memory** template engine, so it does all of its processing in memory which makes it quite fast. It builds a DOM that maps to the HTML of the page and binds the values to those fields which are displayed, and when the values from the server change the parsed fields and pages are updated accordingly. Its caching is also an in-memory caching system, which means that the cache will invalidate with every server restart.

Thymeleaf is a template engine that relies mostly on **attributes**, unlike common engines which rely on **tags** as in JSP or JSF. This makes it testable in the browser directly without requiring a server to parse the special HTML tags, which eases the work between designer and developer, as they can both test the same page.

How it works

Let’s consider this piece of code:

```
<p th:text="'Thymeleaf will display this'">text</p>
```

Here Thymeleaf will process the text inside the `th:text` attribute, and replace the contents of the `<p>` tag with it. Thymeleaf works by replacing the contents of the tags that its attributes are defined on. so the final in the browser output will be:

```
<p>Thymeleaf will display this</p>
```

Notice that the special attributes are now gone, as well as the text “text” which is now replace with the contents of the Thymeleaf attribute.

A more complicated example:

```
<tr th:each="prod : ${prods}">
    <td th:text="${prod.name}">Onions</td>
    <td th:text="${prod.price}">2.41</td>
<tr>
```

Here Thymeleaf will repeat the `<tr>` with the list of products, this is defined by the attribute `th:each`, it will also remove the dummy content in both the `<td>` tags, and replace them with the content that is evaluated from `th:text="${prod.name}"` and `th:text="${prod.price}"`.

Thymeleaf Layout Dialect

This dialect adds JSF-like template hierarchy to the Thymeleaf Engine, which makes it possible that you have templates extending other templates and overriding fragments of those parent templates that we open for extension. This is useful when you have a common layout that you want to apply to all your pages and views, for example a footer or a sidebar or common CSS and JavaScript tags. To start, you need the dependency:

```
<dependency>
    <groupId>nz.net.ultraq.thymeleaf</groupId>
    <artifactId>thymeleaf-layout-dialect</artifactId>
    <version>1.3.1</version>
</dependency>
```

Now you can add your common content in your parent layout page, and then define the part that you want extending templates to substitute with their custom content in a **fragment**, then in extending templates you use this parent template as your **layout-decorator**, then override the **fragment** with your custom content.

For example, say we have a **main.html** which contains a navbar, a sidebar, and a footer, and it has the middle part empty waiting for content to be inserted in it, it then defines this middle part as follows:

```
<div class="container">
    <div layout:fragment="content" class="noborder">
    </div>
</div>
```

Then in the overriding template, for example **index.html**, we use `layout:decorator="main"` at the `<html>` tag, where **main** is the parent template to be extended. Then in **index.html** we do this to override the fragment `content`

```
<div layout:fragment="content">
      <p th:text="${template}">Should not be displayed</p>
</div>
```

This will override the fragment's content with the content in the `div` tag.

For more information, please check this.

Spring Integration

Thymeleaf has a Spring integration project, that eases the integration of **Spring MVC** with Thymeleaf as a template.

1. Add `thymeleaf-spring` dependency to your dependencies

   ```
   <dependency>
         <groupId>org.thymeleaf</groupId>
         <artifactId>thymeleaf-spring4</artifactId>
         <version>2.1.4.RELEASE</version>
    </dependency>
   ```

2. Add this configuration to your Spring servlet configuration

   ```
   <bean id="templateResolver"
        class="org.thymeleaf.templateresolver.ServletContextTemplateResolver">
        <property name="prefix" value="/WEB-INF/templates/" />
        <property name="suffix" value=".html" />
        <property name="templateMode" value="HTML5" />
    </bean>
    <bean id="templateEngine" class="org.thymeleaf.spring4.SpringTemplateEngine">
         <property name="templateResolver" ref="templateResolver" />
         <property name="additionalDialects">
          <set>
             <bean class="nz.net.ultraq.thymeleaf.LayoutDialect" />
          </set>
         </property>
    </bean>
    <bean class="org.thymeleaf.spring4.view.ThymeleafViewResolver">
         <property name="templateEngine" ref="templateEngine" />
    </bean>
   ```

   This configuration will make the Thymeleaf resolver the `ViewResolver` of Spring MVC, the `<property name="templateMode" value="HTML5" />` is particulary important, as it sets the mode of which Thymeleaf should operate, we here specify the mode to be **HTML5**, which means that Thymeleaf should produce valid HTML5 HTML.

Attributes

Thymeleaf is an attribute based template engine - it processes attributes and their values to build its DOM tree.

- `th:text`: this attribute is responsible for displaying text that is evaluated from the expression inside it; it will process the expression and then display the text **HTML-encoded**. For example: `<p th:text="#{home.welcome}">Welcome to our grocery store!</p>`
- `th:utext`: Similar to previous attribute, but this one displays text **unescaped**. For more information check Using Texts.
- `th:attr` : Takes an HTML attribute and sets its value dynamically. Example: `<input type="submit" value="Subscribe me!" th:attr="value=#{subscribe.submit}"/>` The `value` attribute will be set to the value of `#{subscribe.submit}` after processing, replacing the supplied `value="Subscribe me!"`
- `th:value`, `th:action`, `th:href, th:onclick`...etc: Those attributes can be used as a shorthand of the `th:attr` syntax, so the attribute `th:action` is equivalent to `th:attr="action="`.
- `th:attrappend`: This will not replace the attribute value, but only append the value to it, example: `th:attrappend="class=${' ' + cssStyle}"`. For more information check Setting Attribute Values.
- `th:each`: This is the iteration attribute - it is analogous to Java's for-each loop: `for(Object o : list)`, but its syntax is

  ```
  <tr th:each="prod,iterStat : ${prods}" th:class="${iterStat.odd}? 'odd'">
         <td th:text="${prod.name}">Onions</td>
         <td th:text="${prod.price}">2.41</td>
        <td th:text="${prod.inStock}? #{true} : #{false}">yes</td>
    </tr>
  ```

  The `th:each="prod,iterStat : ${prods}"` is equivalent to `for(Product prod : prods)` and the `iterStat` is the status variable of the iteration. It contains information about current iteration like its number, index, total count, etc. The iteration object `prod` can then be accessed in the context of the tag `<th>`, meaning it will only exist within the tag that it's been defined in, for more information check Iteration.
- `th:if`: Evaluates the conditions specified in the attribute and if they are true, the tag is displayed; if not they are not displayed. Example : `th:if="${user.admin}"`
- `th:unless`: Is the opposite of `th:if`, it will display the tag if the value is false, so `th:unless="${user.admin}"` is equal to `th:if="${!(user.admin)}"`.

- `th:switch` and `th:case`: These attributes are used to create a switch statement, `th:switch` will hold the variable to switch on, and `th:case` will evaluate the case statements for this variable. Example:

  ```
  <div th:switch="${user.role}">
        <p th:case="'admin'">User is an administrator</p>
        <p th:case="#{roles.manager}">User is a manager</p>
        <p th:case="*">User is some other thing</p>
    </div>
  ```

  For more information check Conditional Evaluation.

Expressions

Thymeleaf works based on many expressions - it has different expression syntax other than the traditional `${variablename.propertyname}` syntax, namely:

- `#{message.in.proprties.file}` similar to the **i18n** resolver in **JSF**, this expression will look for the value provided in the localization properties files provided to the application. Example: `<p th:text="#{brand.name}">Brand Name</p>`, when using Spring it will use the `MessageSource` of Spring.
- `${variable}`: This is the variables expression, if your expression should evaluate to a variable or you have a variable in your `model` as an attribute, you must use this expression to access it; other expressions are used for different purposes and may not function correctly with variables. Example: `<span th:text="${today}">13 february 2011</span>`
- Thymeleaf provides some predefined variables that can be accessed using the `${#variableName}` syntax and they are:
  1. `#ctx` : the context object.
  2. `#vars`: the context variables.
  3. `#locale` : the context locale.
  4. `#httpServletRequest` : (only in Web Contexts) the `HttpServletRequest` object.
  5. `#httpSession`: The session object of the current session
  6. `#dates` : utility methods for `java.util.Date` objects: formatting, component extraction, etc.
  7. `#calendars` : analogous to `#dates`, but for `java.util.Calendar` objects.
  8. `#numbers` : utility methods for formatting numeric objects.
  9. `#strings` : utility methods for String objects : contains, startsWith, prepending /appending, etc.
  10. `#objects` : utility methods for objects in general.

11. `#bools` : utility methods for boolean evaluation.
12. `#arrays` : utility methods for arrays.
13. `#lists` : utility methods for lists.
14. `#sets` : utility methods for sets.

    Example:

    ```
    <span th:text="${#locale.country}">
    ```

    and

    ```
    <span th:text="${#calendars.format(today,'dd MMMM yyyy')}">13 May 2011</span>
    ```

- `{property}`: This is used the same way as the `${variable}` but works on selected objects, i.e. objects which are set using `th:object` attribute. For example:

```
<div th:object="${session.user}">
    <p>Name: <span th:text="*{firstName}">Sebastian</span>.</p>
    <p>Surname: <span th:text="*{lastName}">Pepper</span>.</p>
    <p>Nationality: <span th:text="*{nationality}">Saturn</span>.</p>
</div>
```

This will access properties on `${session.user}` object directly using the `{...}` syntax, like for `{firstName}`, this is equal to using `${session.user.firstName}` *Note*: The `th:object` is defined only in the context of the tag it’s declared on, this means that it’s not available outside the context of that tag.

- `@{/link/path}`: This will create a link to the path specified relative to the deployment context, so if the application is deployed at context **my-app**, then the generated path will be **/my-app/link/path**.To add get parameters use `@{/link/path(param=value)}` which will generate **/link/path?param=value**For Path variables use: `@{/link/{pathVariable}/path(pathVariable=${variable})}`which will replace the **{pathVariable}** with the value from **${variable}**
- Literals: You can also write some normal literals instead of any expressions,
  - “‘the literal string’”: You can write normal strings between two ‘’ single quotes.
  - “3 + 2”: Normal numeric expressions
  - “false”,”true” and “null”: are evaluted to normal `false`,`true` and `null` expressions.
  - “singleWordToken”: tokens with single words do not need single quotes and can be written as they are.
- `${#fields}`: Spring MVC adds another predefined varable which is `#fields`, it refers to `spring-form` fields and their validation errors, mainly used for error validation.

- `${@beanName.method()}`: Also Spring specific; a bean method call expression. This will call a method on a Spring bean called `beanName`, which will look for the bean in the current Spring context and then execute this method.