Thymeleaf

Getting started with the Standard dialects in 5 minutes

This guide will take you through some of the most important concepts you need to know to understand a Thymeleaf template written in the *Standard* or *SpringStandard* dialects. It is not a substitute for the tutorials – which are much more comprehensive – but it will teach you enough for getting the feel of the technology.

1. Standard dialects?

Thymeleaf is very, very extensible, and it allows you to define your own sets of template attributes (or even tags) with the names you want, evaluating the expressions you want in the syntax you want and applying the logic you want. It's more like a *template engine framework*.

Out of the box, nevertheless, it comes with something called *the standard dialects* (named *Standard* and *SpringStandard*) that define a set of features which should be more than enough for most scenarios. You can identify when these standard dialects are being used in a template because it will contain attributes starting with the `th` prefix, like `<span th:text="...">`.

Note that the *Standard* and the *SpringStandard* dialects are almost identical, except that *SpringStandard* includes specific features for integrating into Spring MVC applications (like, for example, using *Spring Expression Language* for expression evaluation instead of *OGNL*).

Also note we usually refer to features in the Standard dialects when we talk about Thymeleaf without being more specific.

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.

2.1. 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}">
```

The expression above is equivalent (both in OGNL and SpringEL) to:

```
((Book)context.getVariable("book")).getAuthor().getName()
```

But we can find variable expressions in scenarios which not only involve *output*, but more complex processing like *conditionals*, *iteration*…

```
<li th:each="book : ${books}">
```

Here **`${books}`** selects the variable called **`books`** from the context, and evaluates it as an *iterable* to be used at a **`th:each`** loop.

2.2. Selection expressions

```
*{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());
}
```

2.3. 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>
```

Note you can use *variable expressions* inside *message expressions* if you want the message key to be determined by the value of a context variable, or you want to specify variables as parameters:

```
#{${config.adminWelcomeKey}(${session.user.name})}
```

2.4. 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>
```

Link expressions can be relative, in which case no application context will be prefixed to the URL:

Also server-relative (again, no application context to be prefixed):

```
<a th:href="@{~/contents/main}">...</a>
```

And protocol-relative (just like absolute URLs, but browser will use the same HTTP or HTTPS protocol used in the page being displayed):

```
<a th:href="@{//static.mycompany.com/res/initial}">...</a>
```

And of course, Link expressions can be absolute:

```
<a th:href="@{http://www.mycompany.com/main}">...</a>
```

But wait, in an absolute (or protocol-relative) URL… what value does the Thymeleaf Link Expression add? easy: the possibility of URL-rewriting defined by *response filters*: In a Servlet-based web application, for every URL being output (context-relative, relative, absolute…) Thymeleaf will always call the `HttpServletResponse.encodeUrl(...)` mechanism before displaying the URL. Which means that a filter can perform customized URL-rewriting for the application by means of wrapping the `HttpServletResponse` object (a commonly used mechanism).

2.5. 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 are 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>
```

2.6. Literals and operations

A good bunch of types of literals and operations are available:

- Literals:
  - Text literals: `'one text'` , `'Another one!'` ,…
  - Number literals: `0` , `34` , `3.0` , `12.3` ,…
  - Boolean literals: `true` , `false`
  - Null literal: `null`
  - Literal tokens: `one` , `sometext` , `main` ,…
- Text operations:
  - String concatenation: `+`
  - Literal substitutions: `|The name is ${name}|`
- Arithmetic operations:
  - Binary operators: `+` , `-` , `*` , `/` , `%`
  - Minus sign (unary operator): `-`
- Boolean operations:
  - Binary operators: `and` , `or`
  - Boolean negation (unary operator): `!` , `not`
- Comparisons and equality:
  - Comparators: `>` , `<` , `>=` , `<=` (`gt` , `lt` , `ge` , `le`)
  - Equality operators: `==` , `!=` (`eq` , `ne`)
- Conditional operators:
  - If-then: `(if) ? (then)`
  - If-then-else: `(if) ? (then) : (else)`
  - Default: `(value) ?: (defaultvalue)`

2.7. Expression preprocessing

One last thing to know about expressions is there is something called *expression preprocessing*, specified between `__` , which looks like this:

What we are seeing there is a variable expression (**`${sel.code}`**) that will be executed first and which result – let's say, " **`ALL`** " – will be used as a part of the real expression to be executed afterwards, in this case an internationalization one (which would look for the message with key **`selection.ALL`**).

3. Some basic attributes

Let's have a look at a couple of the most basic attributes in the Standard Dialect. Starting with **`th:text`** , which just replaces the body of a tag (notice again the prototyping abilities here):

```
<p th:text="#{msg.welcome}">Welcome everyone!</p>
```

Now **`th:each`** , which repeats the element it's in as many times as specified by the array or list returned by its expression, creating also an inner variable for the iteration element with a syntax equivalent to that of a Java *foreach* expression:

```
<li th:each="book : ${books}" th:text="${book.title}">En las Orillas del Sar</li>
```

Lastly, Thymeleaf includes lots of **`th`** attributes for specific XHTML and HTML5 attributes which just evaluate their expressions and set the value of these attributes to their result. Their names mimic those of the attributes which values they set:

```
<form th:action="@{/createOrder}">

<input type="button" th:value="#{form.submit}" />

<a th:href="@{/admin/users}">
```

4. Want to know more?

Then the *"Using Thymeleaf"* tutorial is what you're looking for!