Thymeleaf base

Standard Expression syntax

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}">
```

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.

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>
```

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>
```

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>
```

```
<!DOCTYPE html>
<html th:fragment="layout (template)">

<head>
  <style>
    * {
      color: red;
    }
  </style>
</head>

<body>
  <th:block th:include="${template}" />
</body>

</html>
```

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org" th:replace="~{examples/layout :: layout(~{::body})}">

<body>
        Hello from body
</body>
</html>
```

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)`

Expression preprocessing

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

```
#{selection.__${sel.code}__}
```

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`).