Thymeleaf notes

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

-capable of processing html,xml,javascript,css and plain text.

-goal-provide highly maintainable way of creating templates.

--process 6 kinds of templates

* Html—no validation check
* Xml—throw exceptions but no validation for dtd/xsd
* Text—allows to use special syntax for templates
* Javascript—works in textual mode
* Css
* Raw

Thymeleaf: is a extremely extensible template engine.

-creating and initializing template engine

ITemplateEngine templateengine = this.application.getTemplateEngine():

-template resolver

ServletContextTemplateResolver templateResolver = new ServletContextTemplateResolver();

-template engine

templateEngine = new TemplateEngine();

templateEngine.setTemplateResolver(templateResolver);

Diff btw jsp and thymeleaf:

Jsp:

* It uses jstl core and spring tags
* It is relatively similar to html
* Uses expression ;languages
* To add css first navigate and then add file
* Should run on server
* Consumes more time to run
* To reload changes must run servers
* It doesn’t completely supports html5
* Internally converted to servlet

Thymeleaf:

* It uses only thymeleaf tags
* It is much similar to html than jsp
* Uses std expression language
* To add css files just add through link and href
* Just double click on file to open
* Consumes less time
* Just refesh to see changes
* Fully supports html5
* Converted into dom object since it saves with html syntax

Spring with thymeleaf:

Thymeleaf allows:

-maps view as templates using @controller class

-Uses spring el instead of ognl

-supports form back binding,validation,error handling,result bindings

-internationalization using messagesource

Views and resolver in spring:

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="viewClass" value="org.springframework.web.servlet.view.JstlView" />

<property name="prefix" value="/WEB-INF/jsps/" />

<property name="suffix" value=".jsp" />

<property name="order" value="2" />

<property name="viewNames" value="\*jsp" />

</bean>

In thymeleaf

<bean class="org.thymeleaf.spring4.view.ThymeleafViewResolver">

<property name="templateEngine" ref="templateEngine" />

<property name="order" value="1" />

<property name="viewNames" value="\*.html,\*.xhtml" />

</bean>

Template resolver in thymeleaf

<bean id="templateResolver"

class="org.thymeleaf.spring4.templateresolver.SpringResourceTemplateResolver">

<property name="suffix" value=".html" />

<property name="templateMode" value="HTML5" />

</bean>

Validations:

a couple of functions in the **#fields** object, the **th:errors** and the **th:errorclass** attributes.

Field errors:

<input type="text" th:field="\*{datePlanted}"

th:class="${#fields.hasErrors('datePlanted')}? fieldError" />

<ul>

<li th:each="err : ${#fields.errors('datePlanted')}" th:text="${err}" />

</ul>

<input type="text" th:field="\*{datePlanted}" />

<p th:if="${#fields.hasErrors('datePlanted')}" th:errors="\*{datePlanted}">Incorrect date</p>

<input type="text" th:field="\*{datePlanted}" />

<p th:if="${#fields.hasErrors('datePlanted')}" th:errors="\*{datePlanted}">Incorrect date</p>

For all errors:

<ul th:if="${#fields.hasErrors('\*')}">

<li th:each="err : ${#fields.errors('\*')}" th:text="${err}">Input is incorrect</li>

</ul>

For global errors:

<ul th:if="${#fields.hasErrors('global')}">

<li th:each="err : ${#fields.errors('global')}" th:text="${err}">Input is incorrect</li>

</ul>

<p th:if="${#fields.hasErrors('global')}" th:errors="\*{global}">Incorrect date</p>

…as well as equivalent **#fields.hasGlobalErrors()** and **#fields.globalErrors()** convenience methods:

<div th:if="${#fields.hasGlobalErrors()}">

<p th:each="err : ${#fields.globalErrors()}" th:text="${err}">...</p>

</div>

Double brackets:

* For variable expressions: **${{...}}**
* For selection expressions: **\*{{...}}**

So, for example, given an Integer-to-String converter that adds commas as a thousands separator, this:

<p th:text="${val}">...</p>

<p th:text="${{val}}">...</p>

…should result in:

<p>1234567890</p>

<p>1,234,567,890</p>

Fragments

This can be a useful componentization tool. For example, it can be used at controllers that execute on AJAX calls, which might return markup fragments of a page that is already loaded at the browser (for updating a select, enabling/disabling buttons…).

<view-state id="detail" view="bookingDetail">

<transition on="updateData">

<render fragments="[//div[@id='data']]"/>

</transition>

</view-state>

Used with ajax calls by creating a bean of vewfactorycreator.

Thymeleaf allows you to process six kinds of templates, each of which is called a Template Mode:

* XML
* Valid XML
* XHTML
* Valid XHTML
* HTML5
* Legacy HTML5

Std dialects:

Ognl-object graph navigation language

 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="...">**.

## S*tandard 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.-variables
* **\*{...}** : Selection expressions.
* **#{...}** : Message (i18n) expressions.
* **@{...}** : Link (URL) expressions.
* **~{...}** : Fragment expressions

Variable expressions (**${...}**) are Spring EL and execute on model attributes, asterisk expressions (**\*{...}**) execute on the form backing bean, hash expressions (**#{...}**) are for internationalization and link expressions (**@{...}**) rewrite URLs..

Syntax:

th:text=”<b>hello</b>”—doesn’t allows the bold it take as less than and greater than signs.

th:utext=”<b>hello</b>”—allows bold world

### *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**).

| **Order** | **Feature** | **Attributes** |
| --- | --- | --- |
| 1 | Fragment inclusion | th:include th:replace |
| 2 | Fragment iteration | th:each |
| 3 | Conditional evaluation | th:if th:unless th:switch th:case |
| 4 | Local variable definition | th:object th:with |
| 5 | General attribute modification | th:attr th:attrprepend th:attrappend |
| 6 | Specific attribute modification | th:value th:href th:src ... |
| 7 | Text (tag body modification) | th:text th:utext |
| 8 | Fragment specification | th:fragment |
| 9 | Fragment removal | th:remove |

Inlining:

Expressions between **[[...]]** are considered expression inlining in Thymeleaf, and in them you can use any kind of expression that would also be valid in a **th:text** attribute.

In order for inlining to work, we must activate it by using the **th:inline** attribute, which has three possible values or modes (**text**, **javascript** and **none**). Let’s try **text**:

<p th:inline="text">Hello, [[${session.user.name}]]!</p>

Script inlining:

Current scripting modes are **javascript** (**th:inline="javascript"**) and **dart** (**th:inline="dart"**).

The first thing we can do with script inlining is writing the value of expressions into our scripts, like:

<script th:inline="javascript">

/\*<![CDATA[\*/

...

var username = /\*[[${session.user.name}]]\*/ 'Sebastian';

...

/\*]]>\*/

</script>

The **/\*[[...]]\*/** syntax, instructs Thymeleaf to evaluate the contained expression.

Adding code:

var x = 23;

/\*[+

var msg = 'Hello, ' + [[${session.user.name}]];

+]\*/

var f = function() {

Removing code:

var x = 23;

/\*[- \*/

var msg = 'This is a non-working template';

/\* -]\*/

var f = function() {

...