What's new in Thymeleaf 2.0

Thymeleaf 2.0 includes a lot of new features, but many of them — in fact, the most important ones in terms of effort— will not be directly visible to those users that did never have the need to create their own dialects for extending Thymeleaf's capabilities.

That is why the explanations you are about to read are not necessarily ordered by relevance, but rather by whether they affect *normal* users or not.

Performance

Thymeleaf 2.0 includes a complete rewrite of its template execution engine and a redesign of most of its internal architecture. This means big improvements in performance since 1.1.

Users do not have to make any changes in their existing templates in order to benefit from these improvements, as these are mainly of internal nature. The only performance modification that could directly affect users is that the **`TemplateEngine.process(...)`** method now allows specifying a **`java.io.Writer`** object as a parameter in order to write the processing results as soon as the DOM tree is processed, without the need to create a **`String`** object containing the whole results in memory (this is especially useful in web scenarios where **`HttpServletResponse`** objects contain one of such *writers*, but it will not affect Spring MVC users, as this optimization is applied automatically in the **`thymeleaf-spring3`** integration package).

A Benchmark has been created for measuring these improvements. You can have a look at the results here.

New `th:switch`/`th:case` attributes in the standard dialects

The new **`th:switch`** attribute works in a very similar way to the **`switch`** structure in the Java language. The expression specified as value is evaluated and its result is compared with the result of evaluating expressions in inner **`th:case`** attributes.

```
<div th:switch="${user.role}">
    <p th:case="'admin'">User is an administrato
    <p th:case="#{roles.manager}">User is a mana
</div>
```

Note that as soon as one **`th:case`** attribute is evaluated as **`true`**, every other **`th:case`** attribute in the same *switch* context is evaluated as **`false`**.

The **`default`** option is specified as **`th:case="*"`**.

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

Added `all-but-first` value to `th:remove`

Prototyping a table usually means to add several tuples (**`<tr>`**) to it; the first of them being the object of iteration (**`th:each`**) and the rest needing to be removed when the template is processed (**`th:remove`**):

```
<table>
    <tr th:each="user : ${users}">
        <td th:text="${user.name}">John Apricot
    </tr>
    <tr th:remove="all">
        <td>Martha Apple</td>
    </tr>
    <tr th:remove="all">
        <td>Frederic Orange</td>
    </tr>
</table>
```

A new value for **`th:remove`**, called **`all-but-first`**, does exactly that and saves some repetitive code:

```
<table th:remove="all-but-first">
    <tr th:each="user : ${users}">
        <td th:text="${user.name}">John Apricot
    </tr>
    <tr>
        <td>Martha Apple</td>
    </tr>
    <tr>
        <td>Frederic Orange</td>
    </tr>
</table>
```

Line number information in errors

The new DOM representation and the generalized template parsing infrastructure (see below) now enable template parsers to add *location* information to DOM nodes, which in practice brings the opportunity to output the number of the line in the template where a processing error has been found:

```
org.thymeleaf.exceptions.TemplateProcessingException:
Exception evaluating SpringEL expression: "#fields.has
org.thymeleaf.spring3.expression.SpelExpressionEvaluat
org.thymeleaf.standard.expression.VariableExpression.e
org.thymeleaf.standard.expression.SimpleExpression.exe
org.thymeleaf.standard.expression.Expression.execute(E
...
```

Simple as it might seem, this was one of the most asked-for enhancements in Thymeleaf, but the old parsing infrastructure just didn't allow it to be implemented... anyway, now the old parsers are gone, so here it is!

DOM Selectors

Thymeleaf 1.1 allowed the inclusion of fragments from other templates by specifying these fragments with an XPath expression between square brackets (`[...]`), like:

```
<div th:include="sometemplate :: [//div[@id='men
</div>
```

Nevertheless, XPath execution heavily relies on the use of the standard DOM API, which Thymeleaf 2.0 replaces by a tailor-made one. Because of this, XPath support has now been replaced by *DOM Selector support* and expressions between square brackets like the one above will now be considered DOM Selector expressions.

And what is a DOM Selector? It is an object of class **`org.thymeleaf.dom.DOMSelector`** that allows you to use a subset of the XPath syntax in order to select a specific region of the original DOM tree. Allowed syntax is as follows:

- **`/x`** means *direct children of the current node with name **x***.
- **`//x`** means *children of the current node with name **x**, at any depth*.
- **`x[@z='v']`** means *elements with name **x** and an attribute called z with value "v"*.
- **`x[@z1='v1' and @z2='v2']`** means *elements with name **x** and attributes **z1** and **z2** with values "v1" and "v2", respectively*.
- **`x[i]`** means *element with name **x** positioned in number **i** among its siblings*.
- **`x[@z='v'][i]`** means *elements with name **x**, attribute **z** with value "v" and positioned in number **i** among its siblings that also match this condition*.

So the following will still be completely valid in 2.0:

```
<div th:include="sometemplate :: [//div[@id='men
</div>
```

Enabled processing of fragmentary templates

Prior to 2.0, Thymeleaf was not adequately designed to process templates that could not be considered *complete documents* from an XML perspective, which limited its scope of applicability in scenarios where it was needed to process only fragments or components of higher-level user interfaces.

The new engine architecture in 2.0 completely removes this restriction so that Thymeleaf can be used to process templates which could be, for example, no longer than a simple <div> block.

Generalized cache infrastructure

Thymeleaf 2.0 completely generalizes the cache infrastructure already present in previous versions. The reason to do this is to give the user complete control over what caches are used or not, and how these should work.

The new **`ICacheManager`** interface defines an extension point that will allow the user to specify his/her own caches (implementations of **`ICache`**) for templates, fragments, externalized messages and expressions.

Also, a standard implementation is included (**`StandardCacheManager`**), providing an easy-to-use API for configuring cache sizes and behaviour without the need of developing custom implementations of the interface.

```
StandardCacheManager manager = new StandardCache
manager.setTemplateCacheMaxSize(5);
manager.setExpressionCacheUseSoftReferences(fals

templateEngine.setCacheManager(manager);
```

New XHTML DTDs for the standard dialects

The new **`th:switch`** and **`th:case`** attributes require a change in the existing Thymeleaf DTDs (those specified with a **`DOCTYPE SYSTEM`** clause), in order to include this new attribute and allow its validation.

New versions of the DTDs for the Standard and SpringStandard dialects have been created, so that the old **`DOCTYPE`** declarations:

```
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/

<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/
```

Can now be substituted by new versions containing the new attribute:

```
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/
```

```
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/

<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/
```

Substituted the standard Java DOM API by a tailor-made DOM representation

Before version 2.0, Thymeleaf used the standard Java DOM API from **`org.w3c.dom.*`** for modelling documents in memory, and template documents using this API where directly exposed to the diverse hierarchies of processors in dialects. This had the advantage of allowing users to work with a well-known API when developing their own dialects and processors.

Unfortunately, the standard DOM API is extremely heavy and complex, and although Thymeleaf only needed to use a percent of all its features, template documents required a lot of memory space and where slower to process than they could be. Besides, several Thymeleaf-specific optimizations existed that could not be applied to DOM processing while using the standard API, so the engine was being penalized even more...

Thymeleaf 2.0 completely removes the standard Java DOM API from its architecture and substitutes it by its own DOM representation (**`org.thymeleaf.dom.*`**). This new DOM representation does not adhere to the DOM standard, although it tries to mirror some parts of its structure and concepts in order to be immediately familiar to users.

The *new DOM* is tailor-made for Thymeleaf, and not only it is much simpler than the old standard, but it also includes as first-class citizens a bunch of important optimizations that allow Thymeleaf to process templates much faster than before, using less resources.

Some advantages of the new DOM are:

- Simpler API, which leads to simpler processors.
- Lighter objects, allowing a lower memory usage.
- Ability to *precompute* the processors to be applied to nodes, so that the sequence of operations to be performed at each node during template processing can be cached along with the DOM tree itself in many scenarios, significantly reducing processing time.
- Allows the generalization of the template parsing infrastructure, so that Thymeleaf 2.0 now uses a SAX parser as default —instead of the old DOM parser, which can still be used but is not default anymore—. This new parser is much faster than the old one, which leads to an additional improvement in performance, even when a template cache

is not being used.

- Allows the generalization of the template result *writing* system (converting DOM trees into the desired output), so that new template formats can be made available. Together with parser generalization, this will effectively allow the generalization of the whole *template mode* infrastructure, as we will see below.

Refactored processor system: the `IProcessor` hierarchy

In Thymeleaf 1.1, *processors* could be classified in two groups: *attribute processors* and *tag processors*. The former type were defined specifically to process DOM elements (*tags*) based on the existence of a specific attribute in these elements. The latter type processed DOM elements (*tags*) just based on their name.

In Thymeleaf 2.0, the new DOM allows a generalization of this schema, so that *attribute processors* and *tag processors* (the latter now called *element processors*) are no longer completely disjoint hierarchies, but instead are children of the new general **`IProcessor`** interface.

And this also means that processors no longer can only apply to DOM elements (*tags*), but instead can apply to any DOM node of any kind, like Text nodes, comments, etc.

Attribute Processors (now extending **`AbstractAttrProcessor`**) and *Element Processors* (now extending **`AbstractElementProcessor`**) are still dealt with in a specific manner at the engine for performance reasons, but they are no longer the only type of processors. For example, the *Text Inliners* that were somewhat hard-coded at the engine in previous versions have now been refactored as *text node processors* (extending **`AbstractTextNodeProcessor`**) executing on Text or CDATA Section nodes, so that their use is consistent with other processors, and also so that they can be easily extended and included into new dialects created by developers.

Finally, the *processor applicability* infrastructure has also been modified in order to simplify it, resulting in the new **`IProcessorMatcher`** hierarchy that substitutes the old **`IApplicability`** features. This new API for specifying when a processor *matches* (*"can be applied to"*) a DOM node is more general —in order to adapt to the new IProcessor needs— and also enables a simpler specification of applicability constraints in processors, resulting in simpler extension code.

Generalized Template Modes: new template reading/writing infrastructure

As already mentioned, another consequence of the new DOM implementation is the ability to manage *generalized template modes*.

What this means is that Thymeleaf no longer can only process templates in one of the six *standard template modes* (**`XML`**, **`VALIDXML`**, **`XHTML`**, **`VALIDXHTML`**, **`HTML5`** and **`LEGACYHTML5`**), but it now allows developers to create their own template modes and use Thymeleaf to process them, just by specifying a *template parser* and a *template writer* for each of them, managed by a structure now known as a *Template Mode Handler* (**`ITemplateModeHandler`**).

Given the fact that a *template parser* (**`ITemplateParser`**) is effectively a means to convert input read by a *resource resolver* into a DOM tree, and also that a *template writer* is effectively a means to convert a DOM tree into the required output format (an XML document, an HTML5 web page, etc.) this new *Template Mode Handler* extension point allows developers to make Thymeleaf process any kind of templates as far as they can be modelled as a DOM tree for processing and then written back to their original format for output.

Out-of-the-box, Thymeleaf 2.0 includes the same standard template modes as before, with the difference that they must now be specified at *template resolver* configuration as Strings instead of using the now-deprecated **`TemplateMode`** enum:

```
ServletContextTemplateResolver templateResolver
// templateResolver.setTemplateMode(TemplateMode
templateResolver.setTemplateMode("HTML5");
```

Minor API modifications

Several other minor modifications have been applied to the diverse engine APIs during the architecture rewrite. Namely:

- Created the **`TemplateProcessingParameters`** class containing all the info available for the processing of the template before its resolution and parsing, and modified the **`ITemplateResolver`** and **`IResourceResolver`** interfaces to use this class as a parameter instead of the more complex **`Arguments`** class.
- Made **`Arguments`** objects include the corresponding **`TemplateResolution`** object created after resolving and parsing each template.

Follow @thymeleaf