Thymeleaf

Tutorial: Extending Thymeleaf

Document version: 20160928 - 28 September 2016
Project version: 3.0.2.RELEASE
Project web site: http://www.thymeleaf.org

1 Some Reasons to Extend Thymeleaf

Thymeleaf is an extremely extensible library. The key to it is that most of its user-oriented features are not directly built into its core, but rather just packaged and componentized into feature sets called *dialects*.

The library offers you two dialects out-of-the-box: the *Standard* and the *SpringStandard* dialects, but you can easily create your own. Let's explore some of the reasons for doing this:

Scenario 1: adding features to the Standard dialects

Say your application uses the *SpringStandard* dialect and that it needs to show an alert text box in blue or red background depending on the user's role (admin or non-admin) from Monday to Saturday, but always in green on Sundays. You can compute this with conditional expressions on your template, but too many conditions could render your code a little bit hard to read...

Solution: create a new attribute called `alertclass` and an attribute processor for it (Java code that will compute the right CSS class), and package it into your own `MyOwnDialect` dialect. Add this dialect to your template engine with the `th` prefix (same as the *SpringStandard* one) and you'll now be able to use `th:alertclass="${user.role}"` !

Scenario 2: view-layer components

Let's say your company uses Thymeleaf extensively, and you want to create a repository of common functionalities (tags and/or attributes) that you can use in several applications without having to copy-paste them from one application to the next. This is, you want to create view-layer components in a similar way to JSPs *taglibs*.

Solution: create a Thymeleaf dialect for each set of related functionalities, and add these dialects to your applications as needed. Note that if the tags or attributes in these dialects make use of externalized (internationalized) messages, you will be able to package these messages along with your dialects (in the shape of *processor messages*) instead of requiring that all of your applications include them in their messages `.properties` files as you would with JSP.

Scenario 3: creating your own template system

Now imagine your are creating a public website that allows users to create their own design templates for showing their content. Of course, you don't want your users to be able to do absolutely everything in their templates, not even all that the Standard Dialect allows (for example, execute OGNL expressions). So you need to offer your users the ability to add to their templates only a very specific set of features that are under your control (like showing a profile photo, a blog entry text, etc).

Solution: create a Thymeleaf dialect with the tags or attributes you want your users to be able to use, like `<mysite:profilePhoto />` or `<mysite:blogentries fromDate="23/4/2011" />` . Then allow your users to create their own templates using these features and just let Thymeleaf execute them, being sure nobody will be doing what they're not allowed to.

2 Dialects and Processors

2.1. Dialects

If you've read the *Using Thymeleaf* tutorial before getting here —which you should have done— you should know that what you've been learning all this time was not exactly *Thymeleaf*, but rather its *Standard Dialect* (or the *SpringStandard Dialect*, if you've also read the *Thymeleaf + Spring* tutorial).

What does that mean? It means that all those `th:x` attributes you learned to use are only a standard, out-of-the-box set of features, but you can define your own set of attributes (or tags) with the names you wish and use them in Thymeleaf to process your templates. *You can define your own dialects.*

Dialects are objects implementing the `org.thymeleaf.dialect.IDialect` interface, which cannot be any simpler:

```
public interface IDialect {

    public String getName();

}
```

The only core requirement of a dialect is to have a name that can be used for its identification. But this alone is of little use, so dialects will normally implement one or several subinterfaces of `IDialect`, depending on what they provide to the Thymeleaf engine:

- `IProcessorDialect` for dialects that provide *processors*.
- `IPreProcessorDialect` for dialects that provide *pre-processors*.
- `IPostProcessorDialect` for dialects that provide *post-processors*.
- `IExpressionObjectDialect` for dialects that provide *expression objects*.
- `IExecutionAttributeDialect` for dialects that provide *execution attributes*.

Processor dialects: `IProcessorDialect`

The `IProcessorDialect` interface looks like this:

```
public interface IProcessorDialect extends IDialect {

    public String getPrefix();
    public int getDialectProcessorPrecedence();
    public Set<IProcessor> getProcessors(final String dialectPrefix);

}
```

Processors are the objects in charge of executing most of the logic in Thymeleaf templates, and possibly the most important Thymeleaf extension artifact. We will cover processors in more detail in next sections.

This dialect only defines three items:

- The *prefix*, which is the prefix or namespace that should be applied *by default* to the elements and attributes matched by the dialect's processors. So a dialect with prefix `th` like e.g. the *Standard Dialect* will be able to define processors matching attributes like `th:text`, `th:if` or `th:whatever` (or `data-th-text`, `data-th-if` and `data-th-whatever` if we prefer *pure HTML5* syntax). Note however that the prefix returned here by a dialect is **only the default one** to be used for that dialect, but such prefix can be changed during template engine configuration. Also note that prefix can be `null` if we want our processors to execute on unprefixed tags/attributes.

- The *dialect precedence* allows the sorting of processors across dialects. Processors define their own *precedence* value, but these processor precedences are considered *relative to dialect precedence*, so every processor in a specific dialect can be configured to be executed before all processors from a different dialect just by setting the correct values for this *dialect precedence*.
- The *processors* are, as its name implies, the set of *processors* provided by the dialect. Note the `getProcessors(...)` method is passed the `dialectPrefix` as an argument in case the dialect has been configured at the Template Engine with a prefix different to the default one. Most probably the `IProcessor` instances will need this information during their initialization.

Pre-processor dialects: `IPreProcessorDialect`

Pre-processors and **post-processors** are different to *processors* in that instead of executing on a single event or on an event model (a fragment of a template), they apply to the entire template execution process as an additional step in the engine's processing chain. Therefore they follow an API completely different to that of processors, much more event-oriented, defined by the lower-level `ITemplateHandler` interface.

In the specific case of the pre-processors, they apply **before** the Thymeleaf engine starts executing processors for a specific template.

The `IPreProcessorDialect` interface looks like:

```
public interface IPreProcessorDialect extends IDialect {

    public int getDialectPreProcessorPrecedence();
    public Set<IPreProcessor> getPreProcessors();

}
```

Which is very similar to the `IProcessorDialect` above –including its own dialect-level precedence for pre-processors– but lacks a *prefix*, as pre-processors don't need it at all (they don't *match* on specific events – instead they handle all of them).

Post-processor dialects: `IPostProcessorDialect`

As stated above, **post-processors** are an additional step in the template execution chain, but this time they execute **after** the Thymeleaf engine has applied all the needed processors. This means post-processors apply just before template output happens (and can therefore modify what is being output).

The `IPostProcessorDialect` interface looks like:

```
public interface IPostProcessorDialect extends IDialect {

    public int getDialectPostProcessorPrecedence();
    public Set<IPostProcessor> getPostProcessors();

}
```

...which is completely analogous to the `IPreProcessorDialect` interface, but of course for post-processors in this case.

Expression Object dialects: `IExpressionObjectDialect`

Dialects implementing this interface provide new *expression objects* or *expression utility objects* that can be used in expressions anywhere in templates, such as the `#strings`, `#numbers`, `#dates`, etc. provided by the Standard Dialect.

The `IExpressionObjectDialect` interface looks like this:

```
public interface IExpressionObjectDialect extends IDialect {

    public IExpressionObjectFactory getExpressionObjectFactory();

}
```

Which, as we can see, does not return the expression objects themselves, but only a *factory*. The reason for this is some *expression objects* might require data from the processing context in order to being built, so it won't be possible to build them until we really are processing the template... and besides, most expressions don't need *expression objects* at all, so it's just better to build them *on demand*, only when they are really needed for specific expressions (and build only those that are needed).

This is the `IExpressionObjectFactory` interface:

```
public interface IExpressionObjectFactory {

    public Map<String,ExpressionObjectDefinition> getObjectDefinitions();

    public Object buildObject(final IProcessingContext processingContext, final String expressionObjectName);

}
```

Execution Attribute dialects: `IExecutionAttributeDialect`

Dialects implementing this interface are allowed to provide *execution attributes*, i.e. objects that are available to every processor being executed during template processing.

For example, the Standard Dialect implements this interface in order to provide to every processor:

- The *Thymeleaf Standard Expression parser* so that Standard Expressions in any attribute can be parsed and executed.
- The *Variable Expression Evaluator* so that `${...}` expressions are executed either in OGNL or SpringEL (depending on whether we are using the Spring integration module or not).
- The *Conversion Service* that performs conversion operations in `${{...}}` expressions.

Note that these objects are not available at the context, so they cannot be used from template expressions. Their availability is limited to implementations of extension points such as processors, pre-processors, etc.

The `IExecutionAttributeDialect` interface is very simple:

```
public interface IExecutionAttributeDialect extends IDialect {

    public Map<String,Object> getExecutionAttributes();

}
```

2.2. Processors

Processors are objects implementing the `org.thymeleaf.processor.IProcessor` interface, and they contain the real logic to be applied on the different parts of a template (which we will represent as **events**, given Thymeleaf is an event-based engine). This interface looks like this:

```
public interface IProcessor {

    public TemplateMode getTemplateMode();
    public int getPrecedence();

}
```

As with dialects, this is a very simple interface that only specified the template mode in which the processor can be applied and its precedence.

But there are several types of *processor*, one for each possible type of event:

- Template start/end
- Element Tags
- Texts
- Comments
- CDATA Sections
- DOCTYPE Clauses
- XML Declarations
- Processing Instructions

And also for **models**: sequences of events representing an *entire element*, i.e. an element with its entire body, including any nested elements or any other kind of artifacts that might appear inside. If the modelled element is a *standalone element*, the model will only contain its corresponding event; but if the modelled element has a body, the model will contain every event from its *open tag* to its *close tag*, both included.

All these types of processors are created by implementing a specific interface, or by extending one of the available *abstract implementations*. All these artifacts conforming the Thymeleaf 3.0 Processor API live at the `org.thymeleaf.processor` package.

Element Processors

Element processors are those that are executed on the *open element* (`IOpenElementTag`) or *standalone element* (`IStandaloneElementTag`) events, normally by means of matching the name of the element (and/or one of its attributes) with a matching configuration specified by the processor. This is what the `IElementProcessor` interface looks like:

```
public interface IElementProcessor extends IProcessor {

    public MatchingElementName getMatchingElementName();
    public MatchingAttributeName getMatchingAttributeName();

}
```

Note however that element processor implementations are not meant to directly implement this interface. Instead, element processors should fall into one of two categories:

- **Element Tag Processors**, implementing the `IElementTagProcessor` interface. These processors execute on *open/standalone tag events only* (no processors can be applied to *close tags*), and have no direct access to the element body.
- **Element Model Processors**, implementing the `IElementModelProcessor` interface. These processors execute on *complete elements*, including their bodies, in the form of `IModel` objects.

We should have a look at each of these interfaces separately:

Element Tag Processors: `IElementTagProcessor`

Element Tag Processors, as explained, execute on the single *open element* or *standalone element* tag that matches its matching configuration (seen in `IElementProcessor`). The interface to be implemented is `IElementTagProcessor`, which looks like this:

```
public interface IElementTagProcessor extends IElementProcessor {

    public void process(
            final ITemplateContext context,
            final IProcessableElementTag tag,
            final IElementTagStructureHandler structureHandler);

}
```

As we can see, besides extending `IElementProcessor` it only specifies a `process(...)` method that will be executed when the *matching configuration* matches (and in the order established by its *precedence*, established at the `IProcessor` superinterface). The `process(...)` signature is quite compact, and follows a pattern found in every Thymeleaf processor interface:

- The `process(...)` method returns `void`. Any actions will be performed via the `structureHandler`.
- The `context` argument contains the context with which the template is being executed: variables, template data, etc.
- The `tag` argument is the event on which the processor is being fired. It contains both the name of the element and its attributes.
- The `structureHandler` is a special object that allows the processor to give instructions to the engine about actions that it should perform as a consequence of the execution of the processor.

Using the `structureHandler`

The `tag` argument passed to `process(...)` is an **immutable** object. So there is no way to, for example, directly modify the attributes of a tag on the `tag` object itself. Instead, the `structureHandler` should be used.

For example, let's see how we would read the value of a specific `tag` attribute, unescape it and keep it in a variable, and then remove the attribute from the tag:

```
// Obtain the attribute value
String attributeValue = tag.getAttributeValue(attributeName);

// Unescape the attribute value
attributeValue =
    EscapedAttributeUtils.unescapeAttribute(context.getTemplateMode(), attributeValue);

// Instruct the structureHandler to remove the attribute from the tag
structureHandler.removeAttribute(attributeName);

... // do something with that attributeValue
```

Note that the code above is only meant to showcase some attribute management concepts – in most processors we won't need to do this "get value + unescape + remove" operation manually as it will all be handled by an extended superclass such as `AbstractAttributeTagProcessor`.

Above we've seen only one of the *operations* offered by the `structureHandler`. There is a *structure handler* for each type of processor in Thymeleaf, and the one for *element tag* processors implements the `IElementTagStructureHandler` interface, which looks like this:

```
public interface IElementTagStructureHandler {

    public void reset();

    public void setLocalVariable(final String name, final Object value);
    public void removeLocalVariable(final String name);

    public void setAttribute(final String attributeName, final String attributeValue);
    public void setAttribute(final String attributeName, final String attributeValue,
                             final AttributeValueQuotes attributeValueQuotes);

    public void replaceAttribute(final AttributeName oldAttributeName,
                                 final String attributeName, final String attributeValue);
    public void replaceAttribute(final AttributeName oldAttributeName,
                                 final String attributeName, final String attributeValue,
                                 final AttributeValueQuotes attributeValueQuotes);

    public void removeAttribute(final String attributeName);
    public void removeAttribute(final String prefix, final String name);
    public void removeAttribute(final AttributeName attributeName);

    public void setSelectionTarget(final Object selectionTarget);

    public void setInliner(final IInliner inliner);

    public void setTemplateData(final TemplateData templateData);

    public void setBody(final String text, final boolean processable);
    public void setBody(final IModel model, final boolean processable);

    public void insertBefore(final IModel model); // cannot be processable
    public void insertImmediatelyAfter(final IModel model, final boolean processable);

    public void replaceWith(final String text, final boolean processable);
    public void replaceWith(final IModel model, final boolean processable);


    public void removeElement();
    public void removeTags();
    public void removeBody();
    public void removeAllButFirstChild();

    public void iterateElement(final String iterVariableName,
                               final String iterStatusVariableName,
                               final Object iteratedObject);

}
```

There we can see all the actions that a processor can ask the template engine to do as a result of its execution. The method names are quite self-explanatory (and there is javadoc for them), but very briefly:

- `setLocalVariable(...)` / `removeLocalVariable(...)` will add a local variable to the template execution. This *local variable* will be accessible during the rest of the execution of the current event, and also during all its *body* (i.e. until its corresponding *close tag*)
- `setAttribute(...)` adds a new attribute to the tag with a specified value (and maybe also type of surrounding quotes). If the attribute already exists, its value will be replaced.
- `replaceAttribute(...)` replaces an existing attribute with a new one, taking its place in the attribute (including its surrounding white space, for example).
- `removeAttribute(...)` removes an attribute from the tag.
- `setSelectionTarget(...)` modifies the object that is to be considered the *selection target*, i.e. the object on which *selection expressions* (`*{...}`) will be executed. In the Standard Dialect, this *selection target* is usually modified by means of the `th:object` attribute, but custom processors can do it too. Note the *selection target* has the same scope as

a local variable, and will therefore be accessible only inside the body of the element being processed.

- `setInliner(...)` modifies the *inliner* to be used for processing all text nodes (`IText` events) appearing in the body of the element being processed. This is the mechanism used by the `th:inline` attribute to enable *inlining* in any of the specified modes (`text` , `javascript` , etc).
- `setTemplateData(...)` modifies the metadata about the template that is actually being processed. When inserting fragments, this allows the engine to know data about the specific fragment being processed, and also the complete stack of fragments being nested.
- `setBody(...)` replaces all the body of the element being processed with the passed text or model (sequence of events = fragment of markup). This is the way e.g. that `th:text` / `th:utext` work. Note that the specified replacement text or model can be set as *processable* or not, depending on whether we want to execute any processors that might be associated with them. In the case of `th:utext="${var}"` , for example, the replacement is set as *non-processable* in order to avoid executing any markup that might be returned by `${var}` as a part of the template.
- `insertBefore(...)` / `insertImmediatelyAfter(...)` allow the specification of a model (fragment of markup) that should appear before or *immediately* after the tag being processed. Note that `insertImmediatelyAfter` means *after the tag being processed* (and therefore as the first part of the element's body) and not *after the entire element that opens here, and closes in a close tag somewhere*.
- `replaceWith(...)` allows the current *element* (entire element) to be replaced with the text or model specified as argument.
- `removeElement()` / `removeTags()` / `removeBody()` / `removeAllButFirstChild()` allow the processor to remove, respectively, the entire element including its body, only the executed tags (open + close) but not the body, only the body but not the wrapping tags, and lastly all the tag's children except the first child element. Note all these options basically mirror the different values that can be used at the `th:remove` attribute.
- `iterateElement(...)` allows the current element (body included) to be iterated as many times as elements exist in the `iteratedObject` (which will usually be a `Collection` , `Map` , `Iterator` or an array). The other two arguments will be used for specifying the names of the variables used for the iterated elements and the status variable.

Abstract implementations for `IElementTagProcessor`

Thymeleaf offers two basic implementations of `IElementTagProcessor` that processors might implement for convenience:

- `org.thymeleaf.processor.element.AbstractElementTagProcessor` , meant for processors that match element events by their element name (i.e. without looking at attributes).
- `org.thymeleaf.processor.element.AbstractAttributeTagProcessor` , meant for processors that match element events by one of their attributes (and optionally also the element name).

Element Model Processors: `IElementModelProcessor`

Element Model Processors execute on the entire elements they match –including their bodies– in the form of an `IModel` object that contains the complete sequence of events that models such element and its contents. The `IElementModelProcessor` is very similar to the one seen above for *tag processors*:

```
public interface IElementModelProcessor extends IElementProcessor {

    public void process(
            final ITemplateContext context,
            final IModel model,
            final IElementModelStructureHandler structureHandler);

}
```

Note how this interface also extends `IElementProcessor` , and how the `process(...)` method it contains follows the same structure as the one in tag processors, replacing `tag` with `model` of course:

- `process(...)` returns `void` . Actions will be performed on `model` or `structureHandler` , not by returning anything.
- `context` contains the execution context: variables, etc.

- `model` is the sequence of events modelling the entire element on which the processor is being executed. This model can be directly modified from the processor.
- `structureHandler` allows instructing the engine to perform actions beyond model modification (e.g. setting local variables).

Reading and modifying the model

The `IModel` object passed as a parameter to the `process()` method is a **mutable** model, so it allows any modifications to be done on it (*models* are mutable, *events* such as *tags* are immutable). For example, we might want to modify it so that we replace every text node from its body with a comment with the same contents:

```
final IModelFactory modelFactory =  context.getModelFactory();

int n = model.size();
while (n-- != 0) {
    final ITemplateEvent event = model.get(n);
    if (event instanceof IText) {
        final IComment comment =
                modelFactory.createComment(((IText)event).getText());
        model.insert(n, comment);
        model.remove(n + 1);
    }
}
```

Note also that the `IModel` interface includes an `accept(IModelVisitor visitor)` method, useful for traversing an entire model looking for specific nodes or relevant data using the *Visitor* pattern.

Using the `structureHandler`

Similarly to *tag processors*, model processors are passed a *structure handler* object that allows them to instruct the engine to take any actions that cannot be done by directly acting on the `IModel model` object itself. The interface these structure handlers implement, much smaller than the one for tag processors, is `IElementModelStructureHandler` :

```
public interface IElementModelStructureHandler {

    public void reset();

    public void setLocalVariable(final String name, final Object value);
    public void removeLocalVariable(final String name);

    public void setSelectionTarget(final Object selectionTarget);

    public void setInliner(final IInliner inliner);

    public void setTemplateData(final TemplateData templateData);

}
```

It's easy to see this is a subset of the one for tag processors. The few methods there work the same way:

- `setLocalVariable(...)` / `removeLocalVariable(...)` for adding/removing local variables that will be available during the model's execution (after the current processor's execution).
- `setSelectionTarget(...)` for modifying the *selection target* applied during the model's execution.
- `setInliner(...)` for setting an inliner.
- `setTemplateData(...)` for setting metadata about the template being processed.

Abstract implementations for `IElementModelProcessor`

Thymeleaf offers two basic implementations of `IElementModelProcessor` that processors might implement for convenience:

- `org.thymeleaf.processor.element.AbstractElementModelProcessor`, meant for processors that match element events by their element name (i.e. without looking at attributes).
- `org.thymeleaf.processor.element.AbstractAttributeModelProcessor`, meant for processors that match element events by one of their attributes (and optionally also the element name).

Template start/end Processors: `ITemplateBoundariesProcessor`

Template Boundaries Processors are a kind of processor that executes on the *template start* and *template end* events fired during template processing. They allow to perform any kind of initialization or disposal of resources at beginning or end of the template processing operation. Note that these events are **only fired for the first-level template**, and not for each of the fragments that might be parsed and/or included into the template being processed.

The `ITemplateBoundariesProcessor` interface looks like this:

```
public interface ITemplateBoundariesProcessor extends IProcessor {

    public void processTemplateStart(
            final ITemplateContext context,
            final ITemplateStart templateStart,
            final ITemplateBoundariesStructureHandler structureHandler);

    public void processTemplateEnd(
            final ITemplateContext context,
            final ITemplateEnd templateEnd,
            final ITemplateBoundariesStructureHandler structureHandler);

}
```

This time the interface offers two `process*(...)` methods, one for the *template start* and another one for the *template end* events. Their signature follows the same pattern as the other `process(...)` methods we saw before, receiving the context, the event object, and the structure handler. Structure handler that, in this case, implements a quite simple `ITemplateBoundariesStructureHandler` interface:

```
public interface ITemplateBoundariesStructureHandler {

    public void reset();

    public void setLocalVariable(final String name, final Object value);
    public void removeLocalVariable(final String name);

    public void setSelectionTarget(final Object selectionTarget);

    public void setInliner(final IInliner inliner);

    public void insert(final String text, final boolean processable);
    public void insert(final IModel model, final boolean processable);

}
```

We can see how, besides the usual methods for managing local variables, selection target and inliner, we can also use the structure handler for inserting text or a model, which in this case will appear at the very beginning or the very end of the result (depending on the event being processed).

Other processors

There are other events for which Thymeleaf 3.0 allows processors to be declared, each of them implementing their corresponding interface:

- **Text** events: interface `ITextProcessor`
- **Comment** events: interface `ICommentProcessor`
- **CDATA Section** events: interface `ICDATASectionProcessor`
- **DOCTYPE Clause** events: interface `IDocTypeProcessor`
- **XML Declaration** events: interface `IXMLDeclarationProcessor`
- **Processing Instruction** events: interface `IProcessingInstructionProcessor`

All of them look pretty much like this (which is the one for text events):

```
public interface ITextProcessor extends IProcessor {

    public void process(
            final ITemplateContext context,
            final IText text,
            final ITextStructureHandler structureHandler);

}
```

Same pattern as all other `process(...)` methods: context, event, structure handler. And these structure handlers are very simple, just like this (again, the one for text events):

```
public interface ITextStructureHandler {

    public void reset();

    public void setText(final CharSequence text);

    public void replaceWith(final IModel model, final boolean processable);

    public void removeText();

}
```

3 Creating our own Dialect

The source code for the examples shown in this and future chapters of this guide can be found in the extraThyme GitHub repository.

3.1. extraThyme: a website for Thymeland's football league

Football is a popular sport in Thymeland[1]. There is a 10-team league going on there each season, and its organizers have just asked us to create a website for it called "extraThyme".

This website will be very simple: just a table with:

- The team names.
- How many matches they won, drew or lost, as well as the total points earned.
- A remark explaining whether their position in the table qualifies them for higher-level competitions next year or else mean their relegation to regional leagues.

Above the league table, a banner will be displaying headlines with the results of recent matches, and also there will be a clearly visible banner that warns users every Sunday that Sundays are match days, and therefore they should be going to the stadium instead of browsing the internet.

extrathyme
Thymeland's football website

Spearmint Caterpillars 1 - 0 Parsley Warriors

League table for May 5, 2016

Today is MATCH DAY!

Team name	Won	Drawn	Lost	Points	Remarks
Spearmint Caterpillars (SPC)	21	10	5	**73**	Classified for the World Champions League
Basil Dragons (BAD)	21	9	6	**72**	Classified for the continental play-offs
Sweet Paprika Savages (SPS)	15	12	9	**57**	Classified for the continental play-offs
Parsley Warriors (PAW)	15	9	12	**54**	
Polar Coriander (PCO)	11	16	9	**49**	
Cinnamon Sailors (CSA)	13	9	14	**48**	
Laurel Troglodytes (LTR)	10	11	15	**41**	
Angry Red Peppers (ARP)	8	8	20	**32**	
Rosemary 75ers (ROS)	7	11	18	**32**	
Saffron Hunters (SHU)	8	7	21	**31**	Relegated to Regional League

extraThyme league table

We will use HTML5, Spring MVC and the SpringStandard dialect for our application, and we will be extending Thymeleaf by creating a `score` dialect that includes:

- A `score:remarkforposition` attribute that outputs internationalized text for the Remarks column in the table. This text should explain whether the team's position in the table qualifies it for the World Champions League, the Continental

Play-Offs, or relegates it to the Regional League.

- A `score:classforposition` attribute that establishes a CSS class for the table rows depending on the team's remarks: blue background for the World Champions League, green for the Continental Play-Offs, and red for relegation.
- A `score:headlines` tag for drawing the yellow box at the top with the results of recent matches. This tag should support an order attribute with values random (for showing a randomly selected match) and `latest` (default, for showing only the last match).
- A `score:match-day-today` attribute that can be added to the league table header in order to (conditionally, if it is Sunday) add a banner warning the user that today is a match day.

Our markup will therefore look like this, making use of both `th` and `score` attributes:

```
<!DOCTYPE html>

<!--/* Note the xmlns:* here are completely optional and only meant to     */-->
<!--/* avoid IDEs from complaining about tags/attributes they may not know */-->
<html xmlns:th="http://www.thymeleaf.org" xmlns:score="http://thymeleafexamples">

  <head>
    <title>extraThyme: Thymeland's football website</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <link rel="stylesheet" type="text/css" media="all"
          href="../../css/extrathyme.css" th:href="@{/css/extrathyme.css}"/>
  </head>

  <body>

    <div>
      <img src="../../images/extrathymelogo.png"
           alt="extraThyme logo" title="extraThyme logo"
           th:src="@{/images/extrathymelogo.png}" th:alt-title="#{title.application}"/>
    </div>

    <score:headlines order="random" />

    <div class="leaguetable">

      <h2 score:match-day-today th:text="#{title.leaguetable(${execInfo.now.time})}">
        League table for 07 July 2011
      </h2>

      <table>
        <thead>
          <tr>
            <th th:text="#{team.name}">Team</th>
            <th th:text="#{team.won}" class="matches">Won</th>
            <th th:text="#{team.drawn}" class="matches">Drawn</th>
            <th th:text="#{team.lost}" class="matches">Lost</th>
            <th th:text="#{team.points}" class="points">Points</th>
            <th th:text="#{team.remarks}">Remarks</th>
          </tr>
        </thead>
        <tbody>
          <tr th:each="t : ${teams}" score:classforposition="${tStat.count}">
            <td th:text="|${t.name} (${t.code})|">The Winners (TWN)</td>
            <td th:text="${t.won}" class="matches">1</td>
            <td th:text="${t.drawn}" class="matches">0</td>
            <td th:text="${t.lost}" class="matches">0</td>
            <td th:text="${t.points}" class="points">3</td>
            <td score:remarkforposition="${tStat.count}">Great winner!</td>
          </tr>
          <!--/*-->
          <tr>
            <td>The First Losers (TFL)</td>
            <td class="matches">0</td>
```

```
            <td class="matches">1</td>
            <td class="matches">0</td>
            <td class="points">1</td>
            <td>Little loser!</td>
          </tr>
          <tr>
            <td>The Last Losers (TLL)</td>
            <td class="matches">0</td>
            <td class="matches">0</td>
            <td class="matches">1</td>
            <td class="points">0</td>
            <td>Big looooooser</td>
          </tr>
          <!--*/-->
        </tbody>
      </table>

    </div>

  </body>

</html>
```

(Note that we've added a second and third rows to our table, surrounded by parser-level comments `<!--/* ... */-->` so that our template shows nicely as a prototype when directly opened in a browser.)

3.2. Changing the CSS class by team position

The first attribute processor we will develop will be `ClassForPositionAttributeTagProcessor`, which we will implement as a subclass of a convenience abstract class provided by Thymeleaf called `AbstractAttributeTagProcessor`.

This abstract class is the base for all tag processors (i.e. processors that act on *tag* events and not *models*) which match (i.e. are selected for execution) based on the existence of a specific attribute in such tag. In this case, `score:classforposition`.

The idea is that we will use this processor for setting a new value to the `class` attribute of the tag `score:classforposition` is living in.

Let's have a look at our code:

```
public class ClassForPositionAttributeTagProcessor extends AbstractAttributeTagProcessor {

    private static final String ATTR_NAME = "classforposition";
    private static final int PRECEDENCE = 10000;


    public ClassForPositionAttributeTagProcessor(final String dialectPrefix) {
        super(
            TemplateMode.HTML, // This processor will apply only to HTML mode
            dialectPrefix,     // Prefix to be applied to name for matching
            null,              // No tag name: match any tag name
            false,             // No prefix to be applied to tag name
            ATTR_NAME,         // Name of the attribute that will be matched
            true,              // Apply dialect prefix to attribute name
            PRECEDENCE,        // Precedence (inside dialect's own precedence)
            true);             // Remove the matched attribute afterwards
    }


    @Override
    protected void doProcess(
            final ITemplateContext context, final IProcessableElementTag tag,
```

```
            final AttributeName attributeName, final String attributeValue,
            final IElementTagStructureHandler structureHandler) {

        final IEngineConfiguration configuration = context.getConfiguration();

        /*
         * Obtain the Thymeleaf Standard Expression parser
         */
        final IStandardExpressionParser parser =
                StandardExpressions.getExpressionParser(configuration);

        /*
         * Parse the attribute value as a Thymeleaf Standard Expression
         */
        final IStandardExpression expression = parser.parseExpression(context, attributeValue);

        /*
         * Execute the expression just parsed
         */
        final Integer position = (Integer) expression.execute(context);

        /*
         * Obtain the remark corresponding to this position in the league table.
         */
        final Remark remark = RemarkUtil.getRemarkForPosition(position);

        /*
         * Select the adequate CSS class for the element.
         */
        final String newValue;
        if (remark == Remark.WORLD_CHAMPIONS_LEAGUE) {
            newValue = "wcl";
        } else if (remark == Remark.CONTINENTAL_PLAYOFFS) {
            newValue = "cpo";
        } else if (remark == Remark.RELEGATION) {
            newValue = "rel";
        } else {
            newValue = null;
        }

        /*
         * Set the new value into the 'class' attribute (maybe appending to an existing value)
         */
        if (newValue != null) {
            if (attributeValue != null) {
                structureHandler.setAttribute("class", attributeValue + " " + newValue);
            } else {
                structureHandler.setAttribute("class", newValue);
            }
        }

    }

}
```

The basic logic flow is easy to see and understand: get the value of the attribute, use it for computing what we need, and finally use the `structureHandler` to instruct the engine about the modifications needed as a result.

It is important to note that we are creating this attribute with the ability of executing expressions written in the Standard Syntax (used by both the *Standard* and the *SpringStandard* dialects). This is, the ability to be set values like `${var}`, `#{messageKey}`, conditionals, etc. See how we use this in our template:

```
<tr th:each="t : ${teams}" score:classforposition="${tStat.count}">
```

In order to evaluate these expressions (also called *Thymeleaf Standard Expressions*) we need to first obtain the Standard Expression Parser, then parse the attribute value, and finally execute the parsed expresion:

```
final IStandardExpressionParser parser =
        StandardExpressions.getExpressionParser(configuration);

final IStandardExpression expression = parser.parseExpression(context, attributeValue);

final Integer position = (Integer) expression.execute(context);
```

It's also interesting the way we use the `structureHandler` to add a new attribute to the host tag (remember the `tag` object is immutable):

```
if (newValue != null) {
    if (attributeValue != null) {
        structureHandler.setAttribute("class", attributeValue + " " + newValue);
    } else {
        structureHandler.setAttribute("class", newValue);
    }
}
```

Last, note that **HTML-escaping texts and attribute is our responsibility**, but in this case we know all the possible values of the `newValue` variable and they require no escaping, so for the sake of simplicity we are skipping that operation.

3.3. Displaying an internationalized remark

The next thing to do is creating an attribute processor able to display the remark text. This will be very similar to the `ClassForPositionAttrProcessor`, but with a couple of important differences:

- We will not be setting a value for an attribute in the host tag, but rather the text body (content) of the tag, in the same way a `th:text` attribute does.
- We need to access the message externalization (internationalization) system from our code so that we can display the text corresponding to the selected locale.

We will be using the same `AbstractAttributeTagProcessor` base class. And this will be our code:

```
public class RemarkForPositionAttributeTagProcessor extends AbstractAttributeTagProcessor {

    private static final String ATTR_NAME = "remarkforposition";
    private static final int PRECEDENCE = 12000;


    public RemarkForPositionAttributeTagProcessor(final String dialectPrefix) {
        super(
            TemplateMode.HTML, // This processor will apply only to HTML mode
            dialectPrefix,     // Prefix to be applied to name for matching
            null,              // No tag name: match any tag name
            false,             // No prefix to be applied to tag name
            ATTR_NAME,         // Name of the attribute that will be matched
            true,              // Apply dialect prefix to attribute name
            PRECEDENCE,        // Precedence (inside dialect's precedence)
            true);             // Remove the matched attribute afterwards
    }


    @Override
    protected void doProcess(
            final ITemplateContext context, final IProcessableElementTag tag,
            final AttributeName attributeName, final String attributeValue,
```

```
            final IElementTagStructureHandler structureHandler) {

        final IEngineConfiguration configuration = context.getConfiguration();

        /*
         * Obtain the Thymeleaf Standard Expression parser
         */
        final IStandardExpressionParser parser =
                StandardExpressions.getExpressionParser(configuration);

        /*
         * Parse the attribute value as a Thymeleaf Standard Expression
         */
        final IStandardExpression expression =
                parser.parseExpression(context, attributeValue);

        /*
         * Execute the expression just parsed
         */
        final Integer position = (Integer) expression.execute(context);

        /*
         * Obtain the remark corresponding to this position in the league table
         */
        final Remark remark = RemarkUtil.getRemarkForPosition(position);

        /*
         * If no remark is to be applied, just set an empty body to this tag
         */
        if (remark == null) {
            structureHandler.setBody("", false); // false == 'non-processable'
            return;
        }

        /*
         * Message should be internationalized, so we ask the engine to resolve
         * the message 'remarks.{REMARK}' (e.g. 'remarks.RELEGATION'). No
         * parameters are needed for this message.
         *
         * Also, we will specify to "use absent representation" so that, if this
         * message entry didn't exist in our resource bundles, an absent-message
         * label will be shown.
         */
        final String i18nMessage =
                context.getMessage(
                        RemarkForPositionAttributeTagProcessor.class,
                        "remarks." + remark.toString(),
                        new Object[0],
                        true);

        /*
         * Set the computed message as the body of the tag, HTML-escaped and
         * non-processable (hence the 'false' argument)
         */
        structureHandler.setBody(HtmlEscape.escapeHtml5(i18nMessage), false);

    }

}
```

Accessing i18n messages

Note that we are accessing the message externalization system with:

```
final String i18nMessage =
        context.getMessage(
                RemarkForPositionAttributeTagProcessor.class,
                "remarks." + remark.toString(),
                new Object[0],
                true);
```

This will call the message resolution mechanism configured at the engine, passing not only the specific key we are interested on and its parameters (none, in this case), but also two other pieces of information:

- The *origin* to be assigned to the message: `RemarkForPositionAttributeTagProcessor.class`
- Whether an *absent message representation* should be used (`true`)

Message resolution is an **extension point** in Thymeleaf (`IMessageResolver` interface), and therefore how these parameters are treated depends on the specific implementation being used. The default implementation in non-Spring-enabled applications (`StandardMessageResolver`) will do the following:

- First look for `.properties` files with the same name as the template file + the locale. So if the template is `/views/main.html` and locale is `gl_ES` , it will look for `/views/main_gl_ES.properties` , then `/views/main_gl.properties` and last `/views/main.properties` .
- If not found, then use the *origin* class (which could have been specified `null`) and look for `.properties` files in classpath with the name of the class specified there (the processor's own class): `classpath:thymeleafexamples/extrathyme/dialects/score/RemarkForPositionAttributeTagProcessor_gl_ES.properties` , etc. This allows the *componentization* or processors and dialects with their whole set of i18n resource bundles in plain old `.jar` files.
- If none of these are found, have a look at the *absent message representation* flag. If `false` , simply return `null` . If `true` , create some kind of text that will allow the developer or user to quickly identify the fact that an i18n resource is missing: `??remarks.rel_gl_ES??` .

(Note that, in Spring-enabled applications, this message resolution mechanism will be replaced by default with Spring's own, based on the `MessageSource` beans declared at the Spring Application Context.)

HTML-escaping content

Also, in this processor we are performing the required HTML-escaping of the content we are setting by using the `HtmlEscape` class from the Unbescape library, used for this purpose throughout Thymeleaf:

```
structureHandler.setBody(HtmlEscape.escapeHtml5(i18nMessage), false);
```

3.4. An element processor for our headlines

The third processor we will write is an element (tag) processor. Note we call this an *element tag processor* in contrast with the two previous processors, which were *attribute tag processors*. The reason is, in this case we want our processor to match (i.e. to be selected for execution) based on the **name of the tag**, not on the name of one of its attributes.

This kind of tag processor has one advantage and also one disadvantage with respect to attribute tag processors:

- Advantage: elements can contain multiple attributes, and so your element processors can receive a richer and more complex set of configuration parameters.
- Disadvantage: custom elements/tags are unknown to browsers, and so if you are developing a web application using custom tags you might have to sacrifice one of the most interesting features of Thymeleaf: the ability to statically display templates as prototypes (*natural templating*).

This processor will extend `AbstractElementTagProcessor`, the base class to be used for tag processors that do not match on a specific attribute:

```
public class HeadlinesElementTagProcessor extends AbstractElementTagProcessor {

    private static final String TAG_NAME = "headlines";
    private static final int PRECEDENCE = 1000;


    private final Random rand = new Random(System.currentTimeMillis());


    public HeadlinesElementTagProcessor(final String dialectPrefix) {
        super(
            TemplateMode.HTML, // This processor will apply only to HTML mode
            dialectPrefix,     // Prefix to be applied to name for matching
            TAG_NAME,          // Tag name: match specifically this tag
            true,              // Apply dialect prefix to tag name
            null,              // No attribute name: will match by tag name
            false,             // No prefix to be applied to attribute name
            PRECEDENCE);       // Precedence (inside dialect's own precedence)
    }


    @Override
    protected void doProcess(
            final ITemplateContext context, final IProcessableElementTag tag,
            final IElementTagStructureHandler structureHandler) {

        /*
         * Obtain the Spring application context.
         */
        final ApplicationContext appCtx = SpringContextUtils.getApplicationContext(context);

        /*
         * Obtain the HeadlineRepository bean from the application context, and ask
         * it for the current list of headlines.
         */
        final HeadlineRepository headlineRepository = appCtx.getBean(HeadlineRepository.class);
        final List<Headline> headlines = headlineRepository.findAllHeadlines();

        /*
         * Read the 'order' attribute from the tag. This optional attribute in our tag
         * will allow us to determine whether we want to show a random headline or
         * only the latest one ('latest' is default).
         */
        final String order = tag.getAttributeValue("order");

        String headlineText = null;
        if (order != null && order.trim().toLowerCase().equals("random")) {
            // Order is random

            final int r = this.rand.nextInt(headlines.size());
            headlineText = headlines.get(r).getText();

        } else {
            // Order is "latest", only the latest headline will be shown

            Collections.sort(headlines);
            headlineText = headlines.get(headlines.size() - 1).getText();

        }

        /*
         * Create the DOM structure that will be substituting our custom tag.
         * The headline will be shown inside a '<div>' tag, and so this must
```

```
         * be created first and then a Text node must be added to it.
         */
        final IModelFactory modelFactory = context.getModelFactory();

        final IModel model = modelFactory.createModel();

        model.add(modelFactory.createOpenElementTag("div", "class", "headlines"));
        model.add(modelFactory.createText(HtmlEscape.escapeHtml5(headlineText)));
        model.add(modelFactory.createCloseElementTag("div"));

        /*
         * Instruct the engine to replace this entire element with the specified model.
         */
        structureHandler.replaceWith(model, false);

    }

}
```

The first interesting part of the code above is showing how to access Spring's `ApplicationContext` in order to obtain one of our beans from it (the `HeadlineRepository`):

```
final ApplicationContext appCtx = SpringContextUtils.getApplicationContext(context);
```

Also, this processor is different to the previous ones in that we will need to *create markup* as a result of its execution: we are going to replace the original `<score:headlines .../>` tag with a `<div>...</div>` fragment, so we will need to make use of the **model factory**.

The Model Factory

The model factory is a special object available to processors (and other structures such as pre-processors, post-processors, etc.) that can create new instances of *events* as *models* (fragments of templates), and also new instances of *models* themselves.

It is therefore the tool for creating new markup, like we can see in the code above:

```
final IModelFactory modelFactory = context.getModelFactory();

final IModel model = modelFactory.createModel();

model.add(modelFactory.createOpenElementTag("div", "class", "headlines"));
model.add(modelFactory.createText(HtmlEscape.escapeHtml5(headlineText)));
model.add(modelFactory.createCloseElementTag("div"));
```

Note how markup events needs to be created *one event at a time*, and how the open and close tags for the same `div` element have to be created separately and in the correct order. This is because models are *sequences of events* and not nodes in a Document Object Model (DOM).

The model factory offers a quite complete set of methods for creating all types of events: tags, texts, DOCTYPEs... and also useful methods for modifying the attributes in a tag (by creating a new `tag` instance, given they are immutable), such as:

```
final IOpenElemenTag newTag = modelFactory.setAttribute(tag, "class", "newvalue");
```

Also, the model factory is able to create `IModel` instances from scratch (like the `modelFactory.createModel()` above), from a single existing event, and also from a piece of markup that we want to convert into its corresponding sequence of events by *parsing* it:

```
final IModel model =
        modelFactory.parse(
                context.getTemplateData(),
                "<div class='headlines'>Some headlines</div>");
```

3.5. A model processor for our “Match Day Today” banner

The last processor we will include in our dialect is of a different nature than the ones we've seen so far: it is a **model processor**, not a *tag processor*.

As already mentioned in a previous section, model processors do not execute on a specific tag event, but on the complete sequence of events (i.e. the *model*) that contains the entire element they are matching.

So if we have a model processor that matches `<p>` tags with attribute `score:matcher`, and a fragment of template such as:

```
<p score:matcher="whatever">
    This is some body
</p>
```

That *model processor* will receive as an argument of its `doProcess()` method an `IModel` containing 3 events: `<p score:matcher="whatever">` (open tag), `\n This is some body\n` (text) and `</p>` (close tag).

So back to our requirements: we wanted a model processor matching a `scrore:match-day-today`, that we can apply to the league table header and make it display, below this header, a banner warning the user that sundays are match days:

```
<h2 score:match-day-today th:text="#{title.leaguetable(${execInfo.now.time})}">
    League table for 07 July 2011
</h2>
```

Note that we don't need a value for this `score:match-day-today` attribute, so we can just ignore it. Our code will like like this:

```
public class MatchDayTodayModelProcessor extends AbstractAttributeModelProcessor {

    private static final String ATTR_NAME = "match-day-today";
    private static final int PRECEDENCE = 100;


    public MatchDayTodayModelProcessor(final String dialectPrefix) {
        super(
            TemplateMode.HTML, // This processor will apply only to HTML mode
            dialectPrefix,     // Prefix to be applied to name for matching
            null,              // No tag name: match any tag name
            false,             // No prefix to be applied to tag name
            ATTR_NAME,         // Name of the attribute that will be matched
            true,              // Apply dialect prefix to attribute name
            PRECEDENCE,        // Precedence (inside dialect's own precedence)
            true);             // Remove the matched attribute afterwards
    }


    protected void doProcess(
            final ITemplateContext context, final IModel model,
            final AttributeName attributeName, final String attributeValue,
            final IElementModelStructureHandler structureHandler) {


        if (!checkPositionInMarkup(context)) {
```

```
            throw new TemplateProcessingException(
                    "The " + ATTR_NAME + " attribute can only be used inside a " +
                    "markup element with class \"leaguetable\"");
        }

        final Calendar now = Calendar.getInstance(context.getLocale());
        final int dayOfWeek = now.get(Calendar.DAY_OF_WEEK);

        // Sundays are Match Days!!
        if (dayOfWeek == Calendar.SUNDAY) {

            // The Model Factory will allow us to create new events
            final IModelFactory modelFactory = context.getModelFactory();

            // We will be adding the "Today is Match Day" banner just after
            // the element we are processing for:
            //
            // <h4 class="matchday">Today is MATCH DAY!</h4>
            //
            model.add(modelFactory.createOpenElementTag("h4", "class", "matchday")); //
            model.add(modelFactory.createText("Today is MATCH DAY!"));
            model.add(modelFactory.createCloseElementTag("h4"));

        }

    }


    private static boolean checkPositionInMarkup(final ITemplateContext context) {

        /*
         * We want to make sure this processor is being applied inside a container tag which has
         * class="leaguetable". So we need to check the second-to-last entry in the element stack
         * (the last entry is the tag being processed itself).
         */

        final List<IProcessableElementTag> elementStack = context.getElementStack();
        if (elementStack.size() < 2) {
            return false;
        }

        final IProcessableElementTag container = elementStack.get(elementStack.size() - 2);
        if (!(container instanceof IOpenElementTag)) {
            return false;
        }

        final String classValue = container.getAttributeValue("class");
        return classValue != null && classValue.equals("leaguetable");

    }


}
```

The first thing to note is that we are performing a check on the position the attribute is being used at: we will only allow it inside a container with `class="leaguetable"`. So our `checkPositionInMarkup(...)` method makes use of the *element stack* in order to know the list of tags that had to be processed in order to process the current one.

Also, regarding the way the new banner element is created (an `<h4>`) notice how what we are doing is modifying the `model` attribute passed as an argument to `doProcess(...)`. No new model object is being created:

```
final IModelFactory modelFactory = context.getModelFactory();

model.add(modelFactory.createOpenElementTag("h4", "class", "matchday")); //
model.add(modelFactory.createText("Today is MATCH DAY!"));
model.add(modelFactory.createCloseElementTag("h4"));
```

3.6. Declaring it all: the Dialect

The last step we need to take in order to complete our dialect is, of course, the dialect class itself.

As seen in a previous section, dialects might implement different interfaces depending on what they provide to the template engine. In this case, our dialect is only providing processors so it will be implementing `IProcessorDialect`.

In fact, we will extend an abstract convenience implementation that will ease the implementation of the interface: `AbstractProcessorDialect`:

```
public class ScoreDialect extends AbstractProcessorDialect {

    private static final String DIALECT_NAME = "Score Dialect";


    public ScoreDialect() {
        // We will set this dialect the same "dialect processor" precedence as
        // the Standard Dialect, so that processor executions can interleave.
        super(DIALECT_NAME, "score", StandardDialect.PROCESSOR_PRECEDENCE);
    }

    /*
     * Two attribute processors are declared: 'classforposition' and
     * 'remarkforposition'. Also one element processor: the 'headlines'
     * tag.
     */
    public Set<IProcessor> getProcessors(final String dialectPrefix) {
        final Set<IProcessor> processors = new HashSet<IProcessor>();
        processors.add(new ClassForPositionAttributeTagProcessor(dialectPrefix));
        processors.add(new RemarkForPositionAttributeTagProcessor(dialectPrefix));
        processors.add(new HeadlinesElementTagProcessor(dialectPrefix));
        processors.add(new MatchDayTodayModelProcessor(dialectPrefix));
        return processors;
    }


}
```

Once our dialect is created, we will need to add it to our Template Engine object in order to use it. This being a Spring-enabled application, we will modify the declared template engine bean:

```
@Bean
public SpringTemplateEngine templateEngine(){
    SpringTemplateEngine templateEngine = new SpringTemplateEngine();
    templateEngine.setTemplateResolver(templateResolver());
    templateEngine.addDialect(new ScoreDialect());
    return templateEngine;
}
```

Note that the `addDialect(...)` call there will add the Score Dialect to the one already configured by default in a `SpringTemplateEngine`: the SpringStandard dialect.

And that’s it! Our dialect is ready to run now, and our league table will display in exactly the way we wanted.

. European football, of course ;-)