Tutorial: Extending Thymeleaf

Document Version: 20110717 – 17 July 2011

Project Version: 1.0.0

Project web site: http://www.thymeleaf.org

CONTENTS

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 SpringStandard dialect, 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 anything 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 3* 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 looks like this:

```
public interface IDialect {

    public String getPrefix();
    public boolean isLenient();

    public Set<IAttrProcessor> getAttrProcessors();
    public Set<ITagProcessor> getTagProcessors();
    public Set<IValueProcessor> getValueProcessors();

    public Set<IDocTypeTranslation> getDocTypeTranslations();
    public Set<IDocTypeResolutionEntry> getDocTypeResolutionEntries();

}
```

Let's see these methods step by step:

First, the *prefix*:

```
    public String getPrefix();
```

This is the prefix that the tags and attributes of your dialect will have, a kind of namespace (although it can be changed when adding dialects to the Template Engine). If you create an attribute named `earth` and your dialect prefix is `planets`, you will write this attribute in your templates as `planets:earth`.

The prefix for both the Standard and SpringStandard Dialects is, obviously, `th`. Prefix can be null so that you can define attribute/tag processors for non-namespaced tags (for example, standard `<p>`, `<div>` or `<table>` tags in XHTML).

Now the *leniency* flag:

```
    public boolean isLenient();
```

A dialect is considered lenient when it allows the existence of attributes or tags in a template which name starts with the specified prefix (e.g.: `<input planets:saturn="..." />`) but no attribute/tag processor is defined in the template for it (there would be no defined behaviour for `planets:saturn`). If this happens, a lenient dialect would simply ignore the attribute/tag.

Both the Standard and the SpringStandard dialects are **not** lenient.

Now, let's have a look at the most important part of the `IDialect` interface, the processors:

```
    public Set<IAttrProcessor> getAttrProcessors();
    public Set<ITagProcessor> getTagProcessors();
    public Set<IValueProcessor> getValueProcessors();
```

There are three kinds of processors:

- *Attribute processors* are objects in charge of processing the dialect's attributes (e.g: `th:text`).
- *Tag processors* are objects in charge of processing the dialect's tags (the Standard Dialect defines no tags).
- *Value processors* are objects in charge of processing values in attributes or tags. These classes are made available to attribute and tag processors by the template engine during their execution.

We will cover processors in more detail in next sections.

More interface methods:

```
    public Set<IDocTypeTranslation> getDocTypeTranslations();
```

This returns the set of *DOCTYPE translations* to be applied. If you remember from the *Using Thymeleaf* tutorial, Thymeleaf can perform a series of DOCTYPE translations that allow you to establish a specific DOCTYPE for your templates and expect this DOCTYPE to be translated into another one in your output.

Last method:

```
    public Set<IDocTypeResolutionEntry> getDocTypeResolutionEntries();
```

This method returns the *DOCTYPE resolution entries* available for the dialect. DOCTYPE resolution entries allow Thymeleaf's XML Parser to locally resolve DTDs linked from your templates (thus avoiding remote HTTP requests for retrieving these DTDs).

Thymeleaf makes most standard XHTML DTDs already available to your dialects by implementing the abstract class `org.thymeleaf.dialect.AbstractXHTMLEnabledDialect`, but you can always add your own ones for your own template DTDs.

2.2 Attribute Processors

Attribute processors implement the `org.thymeleaf.processor.attr.IAttrProcessor` interface which looks like this:

```
public interface IAttrProcessor extends Comparable<IAttrProcessor> {
    public Set<AttrApplicability> getAttributeApplicabilities();

    public Integer getPrecedence();

    public Set<Class<? extends IValueProcessor>> getValueProcessorDependencies();

    public AttrProcessResult process(final Arguments arguments,
            final TemplateResolution templateResolution, final Document document,
            final Element element, final Attr attribute);

}
```

This interface allows the attribute processor to specify the following data:

- The *applicability* of this attribute processor, specified by a set of `AttrApplicability` objects. By means of these objects, an attribute processor can define the name of the attribute/s that it will be applied to, and even select if it is going to be applied only when certain attributes exist or have a specific value in the same tag.
- The *precedence* of this processor. When several attributes are present in the same tag, precedence establishes the order In which processors are executed (smallest goes first).
- The *value processor dependencies*. An attribute processor can make use of several value processors, and specifying these dependencies here allows Thymeleaf to validate dialects on startup making sure all the required value processors are present.

The configuration, context, local variables, the resolved (and parsed) template and the DOM attribute object being processed are received as arguments by the `process(...)` method. This is the method that is called every time the engine finds an attribute corresponding to this processor's applicability.

Note that Thymeleaf includes a hierarchy of abstract convenience implementations of `IAttrProcessor` (all of them extending from `AbstractAttrProcessor`) that allow you to create custom attribute processors easily and writing less code than what you'd need if directly implementing the `IAttrProcessor` interface.

2.3 Tag Processors

Tag processors are the tag equivalents to attribute processors, implementing the `org.thymeteleaf.processor.tag.ITagProcessor` interface:

```
public interface ITagProcessor {

    public Set<TagApplicability> getTagApplicabilities();

    public Set<Class<? extends IValueProcessor>> getValueProcessorDependencies();

    public TagProcessResult process(final Arguments arguments,
            final TemplateResolution templateResolution, final Document document, final Element element);

}
```

This interface provides the following data:

- The *applicability* of this tag processor, specified by a set of TagApplicability objects. By means of these objects, a tag processor can define the name of the tag/s that it will be applied to, and even select if it is going to be applied only when certain attributes exist or have a specific value in the tag.
- The *value processor dependencies*. Just like attribute processors, tag processors can

make use of several value processors, and specifying these dependencies here allows Thymeleaf to validate dialects on startup making sure all the required value processors are present.

Equivalently to attribute processors, the `process(...)` method receives data such as configuration, context, resolved template data, and the DOM element, and is the method that is called every time the engine finds a tag corresponding to this processor's applicability.

Note that Thymeleaf includes a hierarchy of abstract convenience implementations of `ITagProcessor` (all of them extending from `AbstractTagProcessor`) that allow you to create custom attribute processors easily and writing less code than what you'd need if directly implementing the `ITagProcessor` interface.

2.4 Value Processors

Value processors do not directly apply on XML elements like tags and attributes. Instead, they are objects specialized in processing values coming from these tags and attributes. A Value Processor object will be used by one or (usually) many attribute or tag processors.

For example, the Thymeleaf Standard dialect has a lot of Attribute Processors, but only five Value Procesors:

- `StandardLinkValueProcessor`: for processing link values (`@{x(y='z')}`)
- `StandardLiteralValueProcessor`: for literal values (`'xyz'`)
- `StandardMessageValueProcessor`: for internationalization message values (`#{x}`)
- `StandardVariableValueProcessor`: for variable values (`${x.y.z()}`)
- `StandardValueProcessor`: taking care of conditional values, and automatically delegating on one of the previous four value processors depending on the type of values to be processed.

These five value processors are used by `th:text`, `th:utext`, `th:alt`, etc... (in fact only the fifth, which in turn delegates to the rest of them). And of course attribute processors declare these value processors as dependencies in their `getValueProcessorDependencies()` methods.

Let's have a look at the `org.thymeleaf.processor.value.IValueProcessor` interface:

```
public interface IValueProcessor {

    public String getName();

    public Set<Class<? extends IValueProcessor>> getValueProcessorDependencies();

}
```

And that's all. At their most simple, value processors only need to declare their name (which will be useful for identifying them in logs) and their dependencies on other value processors.

Let's have a look now at a value processor interface from the Thymeleaf Standard Dialect, for example the `IStandardVariableValueProcessor`:

```
public interface IStandardVariableValueProcessor extends IValueProcessor {

    public Object getVariableValue(final Arguments arguments, final VarValue varValue);

}
```

As you can see, the real method that executes the value processor (`getVariableValue(...)`, in this case) is not in the `IValueProcessor` interface, but rather in its subclasses/subinterfaces. This is so because each value processor will have very diverse needs and offer diverse features, and so the parameters passed to its methods will vary consequently.

3 Putting it all to work

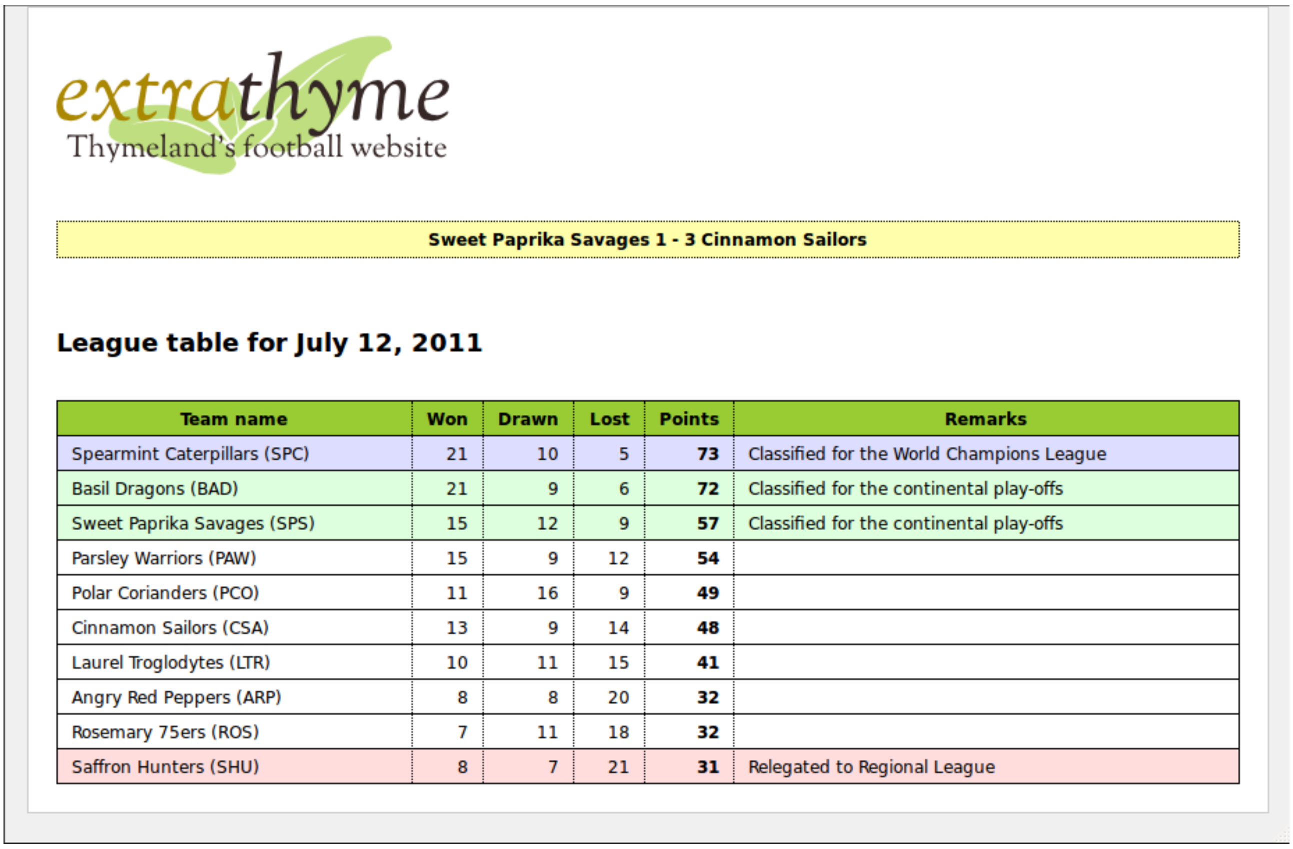
3.1 Extrathyme: a website for Thymeland's football league

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 would 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.

Also, above the league table, a banner will be displaying headlines with the results of recent matches.



Sweet Paprika Savages 1 - 3 Cinnamon Sailors

League table for July 12, 2011

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 Corianders (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

1 European football, of course ;-)

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 an 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).

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

```
<!DOCTYPE html>

<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 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 th:remove="all">
            <td>The Losers (TLS)</td>
            <td class="matches">0</td>
            <td class="matches">0</td>
            <td class="matches">1</td>
            <td class="points">0</td>
            <td>Big loooooser</td>
          </tr>
        </tbody>
      </table>

    </div>

  </body>

</html>
```

(Note that we've added a second row to our table with `th:remove="all"` 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 `ClassForPositionAttrProcessor`, which we will implement as a subclass of a convenience abstract class provided by Thymeleaf called `AbstractAttributeModifierAttrProcessor`.

This abstract class is already oriented to creating attribute processors that set or modify the value of attributes in their host tags, which is exactly what we need (we will set a value to the `<tr>`'s `class` attribute.

Let's have a look at our code:

```
public class ClassForPositionAttrProcessor
        extends AbstractAttributeModifierAttrProcessor {

    public ClassForPositionAttrProcessor() {
        super();
    }

    public Set<AttrApplicability> getAttributeApplicabilities() {
        return AttrApplicability.createSetForAttrName("classforposition");
    }

    public Integer getPrecedence() {
        return Integer.valueOf(12000);
    }

    @Override
    public Set<Class<? extends IValueProcessor>> getValueProcessorDependencies() {
        final Set<Class<? extends IValueProcessor>> dependencies =
            new HashSet<Class<? extends IValueProcessor>>();
        dependencies.add(StandardValueProcessor.class);
        return dependencies;
    }

    @Override
    protected Map<String, String> getNewAttributeValues(final Arguments arguments,
            final TemplateResolution templateResolution, final Document document,
            final Element element, final Attr attribute, final String attributeName,
            final String attributeValue) {

        /*
         * Obtain the value processor, required for executing the expression specified
         * as attribute value.
         */
        final StandardValueProcessor valueProcessor =
            arguments.getConfiguration().getValueProcessorByClass(this, StandardValueProcessor.class);

        /*
         * Parse the attribute value into a Value object. Value objects are objectual
         * representations of expressions, ready to be executed but not containing any
         * specific values yet.
         */
        final Value positionValue =
            StandardSyntax.parseValue(attributeValue, valueProcessor, arguments, templateResolution);

        /*
         * Execute the previously parsed expression (Value object) by specifying the values
         * that should be applied (variables coming from the context contained in the
         * execution arguments).
         */
        final Integer position =
            (Integer) valueProcessor.getValue(arguments, templateResolution, positionValue);

        /*
         * Obtain the remark corresponding to this position in the league table.
         */
        final Remark remark = RemarkUtil.getRemarkForPosition(position);
        /*
         * Apply the corresponding CSS class to the element.
         */
        final Map<String,String> values = new HashMap<String, String>();
        if (remark != null) {
            switch (remark) {
                case WORLD_CHAMPIONS_LEAGUE:
                    values.put("class", "wcl");
                    break;
                case CONTINENTAL_PLAYOFFS:
                    values.put("class", "cpo");
                    break;
                case RELEGATION:
                    values.put("class", "rel");
                    break;
            }
        }

        return values;

    }

    @Override
    protected ModificationType getModificationType(final Arguments arguments,
            final TemplateResolution templateResolution, final Document document,
            final Element element, final Attr attribute, final String attributeName,
            final String attributeValue, final String newAttributeName) {
        // Just in case there already is a value set for the 'class' attribute in the
        // tag, we will append our new value (using a whitespace separator) instead
        // of simply substituting it.
        return ModificationType.APPEND_WITH_SPACE;
    }

    @Override
    protected boolean removeAttributeIfEmpty(final Arguments arguments,
            final TemplateResolution templateResolution, final Document document,
            final Element element, final Attr attribute, final String attributeName,
            final String attributeValue, final String newAttributeName) {
        // If the resulting 'class' attribute is empty, do not show it at all.
        return true;
    }

}
```

As you can see, in this case the convenience abstract class we are using abstracts from us any direct modification on the DOM object tree, and instead we just have to create and return a Map with all the new attribute values to be set in the tag.

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 Standard Syntax expressions we need to use a value processor coming from the *SpringStandard* dialect, specifically a `StandardValueProcessor`. That is the reason why we declare it in the `getValueDependencies()` method, and are later able to use it from our `getNewAttributeValues(...)` method.

> Note that this, of course, makes our dialect dependent on the SpringStandard dialect itself –because it uses one of its IValueProcessor implementations– but in this case it is something we can afford, as we don't plan to use this dialect in non-Spring MVC applications.

3.3 Displaying an internationalized remark

The next thing to do is 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.

This time we will be using a different convenience abstract class –one especially designed for setting the tag's text content–, `AbstractTextChildModifierAttrProcessor`. And this will be our code:

```
public class RemarkForPositionAttrProcessor
        extends AbstractTextChildModifierAttrProcessor {


    public RemarkForPositionAttrProcessor() {
        super();
    }

    public Set<AttrApplicability> getAttributeApplicabilities() {
        return AttrApplicability.createSetForAttrName("remarkforposition");
    }

    public Integer getPrecedence() {
        return Integer.valueOf(12000);
    }

    @Override
    public Set<Class<? extends IValueProcessor>> getValueProcessorDependencies() {
        final Set<Class<? extends IValueProcessor>> dependencies =
            new HashSet<Class<? extends IValueProcessor>>();
        dependencies.add(StandardValueProcessor.class);
        return dependencies;
    }

    @Override
    protected String getText(final Arguments arguments,
            final TemplateResolution templateResolution, final Document document,
            final Element element, final Attr attribute, final String attributeName,
            final String attributeValue) {

        /*
         * Obtain the value processor, required for executing the expression specified
         * as attribute value.
         */
        final StandardValueProcessor valueProcessor =
            arguments.getConfiguration().getValueProcessorByClass(this, StandardValueProcessor.class);

        /*
         * Parse the attribute value into a Value object. Value objects are objectual
         * representations of expressions, ready to be executed but not containing any
         * specific values yet.
         */
        final Value positionValue =
            StandardSyntax.parseValue(attributeValue, valueProcessor, arguments, templateResolution);

        /*
         * Execute the previously parsed expression (Value object) by specifying the values
         * that should be applied (variables coming from the context contained in the
         * execution arguments).
         */
        final Integer position =
            (Integer) valueProcessor.getValue(arguments, templateResolution, positionValue);

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

        /*
         * If no remark is to be applied, just return an empty message
         */
        if (remark == null) {
            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.
         */
        return getMessage(arguments, templateResolution, "remarks." + remark.toString(), new Object[0]);

    }

}
```

Note that we are accessing the message externalization system with:

```
return getMessage(arguments, templateResolution, "remarks." + remark.toString(), new Object[0]);
```

But this in fact is not the only way. The `AbstractAttrProcessor` class offers three methods for obtaining externalized messages from attribute processors. The first two make a difference between *template messages* and *processor messages*:

```
protected String getMessageForTemplate(
        final Arguments arguments, final TemplateResolution templateResolution,
        final String messageKey, final Object[] messageParameters);

protected String getMessageForProcessor(
        final Arguments arguments, final String messageKey, final Object[] messageParameters);
```

`getMessageForTemplate(...)` uses the Template Engine's currently registered externalization mechanisms to look for the desired message. For example:

- In a Spring application, we will probably be using specific Message Resolvers that query the Spring `MessageSource` objects registered for the application.
- When not in a Spring application, we will probably be using Thymeleaf's Standard Message Resolver that looks for `.properties` files with the same name as the template being processed.

`getMessageForProcessor(...)` uses a message resolution mechanism created for allowing the componentization -or, if you prefer, encapsulation- of dialects. This mechanism consists in allowing tag and attribute processors to specify their own messages, whichever the application their dialects are used on. These are read from `.properties` files with the same name and living in the same package as the processor class (or any of its superclasses). For example, the `thymeleafexamples.extrathyme.dialects.score` package in our example could contain:

- `RemarkForPositionAttrProcessor.java`: the attribute processor.
- `RemarkForPositionAttrProcessor_en_GB.properties`: externalized messages for English (UK) language.
- `RemarkForPositionAttrProcessor_en.properties`: externalized messages for English (rest of countries) language.
- `RemarkForPositionAttrProcessor.properties`: default externalized messages.

Finally, there is a third method, the one we used in our code:

```
protected String getMessage(
        final Arguments arguments, final TemplateResolution templateResolution,
        final String messageKey, final Object[] messageParameters);
```

This getMessage(...) acts as a combination of the other two: first it tries to resolve the required message as a template message (defined in the application messages files) and if it doesn't exist tries to resolve it as a processor message. This way, applications can override –if needed– any messages established by its dialects.

3.4 A TAG PROCESSOR FOR OUR HEADLINES

The third and last processor we will have to write is a tag processor. As their name implies, tag processors are executed for tags instead of attributes, and they have one advantage and also one disadvantage with respect to attributes:

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

Our tag processor will implement the `org.thymeleaf.processor.tag.ITagProcessor`, but as we did with our attribute processors, instead of implementing it directly, we will use an abstract convenience implementation as a base: `AbstractMarkupSubstitutionTagProcessor`. This is a base tag processor that simply expects you to generate the DOM nodes that will substitute it (the host tag) when the template is processed.

And this is our code:

```java
public class HeadlinesTagProcessor extends AbstractMarkupSubstitutionTagProcessor {

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

    public HeadlinesTagProcessor() {
        super();
    }

    public Set<TagApplicability> getTagApplicabilities() {
        return TagApplicability.createSetForTagName("headlines");
    }

    @Override
    protected List<Node> getMarkupSubstitutes(final Arguments arguments,
            final TemplateResolution templateResolution, final Document document,
            final Element element) {

        /*
         * Obtain the Spring application context. Being a SpringMVC-based
         * application, we know that the IContext implementation being
         * used is SpringWebContext, and so we can directly cast and ask it
         * to return the AppCtx.
         */
        final ApplicationContext appCtx =
            ((SpringWebContext)arguments.getContext()).getApplicationContext();

        /*
         * 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 = element.getAttribute("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 Element container = document.createElement("div");
        container.setAttribute("class", "headlines");

        final Text text = document.createTextNode(headlineText);
        container.appendChild(text);

        /*
         * The abstract IAttrProcessor implementation we are using defines
         * that a list of nodes will be returned, and that these nodes
         * will substitute the tag we are processing.
         */
        final List<Node> nodes = new ArrayList<Node>();
        nodes.add(container);

        return nodes;

    }

}
```

Not much new to see here, except for the fact that we are accessing Spring's `ApplicationContext` in order to obtain one of our beans from it (the `HeadlineRepository`).

Note also how we can access the custom tag's `order` attribute as we would with any other DOM element:

```
final String order = element.getAttribute("order");
```

3.5 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.

Dialect classes must implement the `org.thymeleaf.dialect.IDialect` interface, but again we will here use an abstract convenience implementation that allows us to only implement the methods we need, returning a default (empty) value for the rest of them.

Here's the code, quite easy to follow by now:

```
public class ScoreDialect extends AbstractDialect {

    /*
     * Default prefix: this is the prefix that will be used for this dialect
     * unless a different one is specified when adding the dialect to
     * the Template Engine.
     */
    public String getPrefix() {
        return "score";
    }
    /*
     * Non-lenient: if a tag or attribute with its prefix ('score') appears on
     * the template and there is no value or tag/attribute processor
     * associated with it, an exception is thrown.
     */
    public boolean isLenient() {
        return false;
    }

    /*
     * Two attribute processors are declared: 'classforposition' and
     * 'remarkforposition'. Both of them make use of the SpringStandard
     * Dialect's value processors for evaluating Spring EL expressions,
     * so these value processors must also be declared at
     * 'getValueProcessors()'.
     */
    @Override
    public Set<IAttrProcessor> getAttrProcessors() {
        final Set<IAttrProcessor> attrProcessors = new HashSet<IAttrProcessor>();
        attrProcessors.add(new ClassForPositionAttrProcessor());
        attrProcessors.add(new RemarkForPositionAttrProcessor());
        return attrProcessors;
    }

    /*
     * Only one tag processor: the 'headlines' tag.
     */
    @Override
    public Set<ITagProcessor> getTagProcessors() {
        final Set<ITagProcessor> tagProcessors = new HashSet<ITagProcessor>();
        tagProcessors.add(new HeadlinesTagProcessor());
        return tagProcessors;
    }

    /*
     * Some of the processors declared for the dialect need to make use of
     * expression evaluation (for example, Spring EL expressions declared
     * with ${...}). For doing so, this dialect needs to declare the set
     * of value processors coming from the SpringStandard Dialect.
     */
    @Override
    public Set<IValueProcessor> getValueProcessors() {
        return SpringStandardDialect.createSpringStandardValueProcessorsSet();
    }

}
```

Once our dialect is created, we will need to declare it for use from our Template Engine. Let's see how we'd configure this in our Spring bean configuration files:

```
<bean id="templateEngine"
      class="org.thymeleaf.spring3.SpringTemplateEngine">
  <property name="templateResolver" ref="templateResolver" />
  <property name="dialects">
    <set>
      <bean class="org.thymeleaf.spring3.dialect.SpringStandardDialect" />
      <bean class="thymeleafexamples.extrathyme.dialects.score.ScoreDialect" />
    </set>
  </property>
</bean>
```

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