



# TUTORIAL: EXTENDING THYMELEAF

**Document Version:** 20120422 – 22 April 2012

**Project Version:** 2.0.6

**Project web site:** <http://www.thymeleaf.org>

# CONTENTS

<b>1</b>	<b>Some reasons to extend Thymeleaf.....</b>	<b>3</b>
<b>2</b>	<b>Dialects and Processors.....</b>	<b>4</b>
2.1	Dialects.....	4
2.2	Processors.....	5
<b>3</b>	<b>Template modes.....</b>	<b>8</b>
<b>4</b>	<b>Creating our own dialect.....</b>	<b>9</b>
4.1	Extrathyme: a website for Thymeland's football league.....	9
4.2	Changing the CSS class by team position.....	11
4.3	Displaying an internationalized remark.....	12
4.4	An element processor for our headlines.....	14
4.5	Declaring it all: the dialect.....	16

# 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 you 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<IProcessor> getProcessors();  
    public Map<String, Object> getExecutionAttributes();  
  
    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

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<IProcessor> getProcessors();
```

Processors are the objects in charge of executing on DOM nodes and performing changes on it. We will cover processors in more detail in next sections.

*Execution attributes* are objects that are contributed by the dialect to the execution arguments during the processing of templates. These are objects –usually utility objects– that will be made available to processors during their execution. Note that these objects will not appear at the variable context, and will be only visible internally.

```
public Map<String,Object> getExecutionAttributes();
```

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 PROCESSORS

Processors are objects implementing the `org.thymeleaf.processor.IProcessor` interface, and they contain the real logic to be applied on DOM nodes. This interface looks like this:

```
public interface IProcessor extends Comparable<IProcessor> {
    public IProcessorMatcher<? extends Node> getMatcher();
    public ProcessorResult process(final Arguments arguments,
                                  final ProcessorMatchingContext processorMatchingContext, final Node node);
}
```

First thing we can see is that it extends `Comparable` – and that is the way *precedence* is established. If a processor is considered to be sorted *before* another one, this means it has more precedence, and therefore will be executed before the latter if they both apply to the same node.

Now for the methods. A processor's *matcher* establishes when a processor is applicable to a DOM node:

```
public IProcessorMatcher<? extends Node> getMatcher();
```

Matcher objects will examine the node's type, name and/or attributes –if it is an *Element* DOM node– or whichever other node features required to determine processor applicability. Thymeleaf comes with a predefined set of `IProcessorMatcher` implementations so that you do not have to perform usual tasks like matching an element –tag– by its name or one of its attributes.

Finally, the method that does the real work:

```
public ProcessorResult process(final Arguments arguments,
                             final ProcessorMatchingContext processorMatchingContext, final Node node);
```

`process(...)` takes three parameters:

1. The execution arguments. An `org.thymeleaf.Arguments` object containing context, local variables, template resolution information and some other bits of data useful for DOM processing.
2. The processor matching context, containing information about the conditions in which the processor being executed was actually matched.

The problem is that the same processor class can be included in several dialects executing at a time in the same template engine –probably with different configurations–, but these dialects might use different prefixes. If so, how can we know the specific dialect for which the processor is being executed? That is the info this `ProcessorMatchingContext` object provides.

3. The node that the processor will be executed on. Note that processors are applied on a specific node, but nothing stops them from modifying any other parts of the DOM tree.

Thymeleaf offers an abstract utility class to be extended for creating processors: `org.thymeleaf.processor.AbstractProcessor`. This class takes care of implementing the `Comparable` interface based on the specified precedence and defines the standard mechanisms for obtaining externalized/internationalized messages:

```
public abstract class AbstractProcessor implements IProcessor {

    /* Try to resolve a message first as template message, then if not found as processor message */
    protected String getMessage(
        final Arguments arguments, final String messageKey, final Object[] messageParameters) {...}

    /* Try to resolve a message as a template message */
    protected String getMessageForTemplate(
        final Arguments arguments, final String messageKey, final Object[] messageParameters) {...}

    /* Try to resolve a message as a processor message */
    protected String getMessageForProcessor(
        final Arguments arguments, final String messageKey, final Object[] messageParameters) {...}

    public abstract int getPrecedence();

    ...
}
```

## Special kinds of processors

Although processors can execute on any node in the DOM tree, there are two specific kinds of processors that can benefit from performance improvements inside the Thymeleaf execution engine: *attribute processors* and *element processors*.

### *Attribute Processors*

Those processors (implementations of `IProcessor`) which `getMatcher()` method returns a matcher implementing the `org.thymeleaf.processor.attr.IAttributeNameProcessorMatcher` interface are considered “*attribute processors*”.

Because of the type of matchers they define, these processors are triggered when a DOM element (usually an XML/XHTML/HTML5 tag) contains an attribute with a specific name. For example, most processors in the *Standard Dialects* act like this, defining matchers for attributes like `th:text`, `th:each`, `th:if`, etc.

For the sake of simplicity, Thymeleaf offers an utility abstract class from which attribute processors can extend: `org.thymeleaf.processor.attr.AbstractAttrProcessor`. This class already returns as matcher an implementation of `IAttributeNameProcessorMatcher` and makes it easier to create this kind of processors.

### *Element Processors*

Those processors (implementations of `IProcessor`) which `getMatcher()` method returns a matcher implementing the `org.thymeleaf.processor.element.IElementNameProcessorMatcher` interface are considered “*element processors*”.

Note that the DOM jargon calls “*element*” to what we usually call “*tag*” in an XML/XHTML/HTML5 document. Thymeleaf prefers to use the word *element* in order to be more general, because template modes might be defined that work on documents that do not have an XML-like structure.

Because of the type of matchers they define, these processors are triggered when a DOM element has a specific name.

The *Standard Dialects* define no element processors.

For the sake of simplicity, Thymeleaf offers an utility abstract class from which element processors can extend: `org.thymeleaf.processor.element.AbstractElementProcessor`. This class already returns as matcher an implementation of `IElementNameProcessorMatcher` and makes it easier to create this kind of processors.

## 3 TEMPLATE MODES

Probably the most powerful extension point in Thymeleaf, template modes define in fact *what can be considered “a template”*. Creating custom template modes allows us to use Thymeleaf for processing templates in formats different to the XML / XHTML / HTML5 that are available out-of-the-box.

Template Modes are defined by their *handlers*. These are objects implementing the `org.thymeleaf.templatemode.ITemplateModeHandler` interface:

```
public interface ITemplateModeHandler {  
    public String getTemplateName();  
    public ITemplateParser getTemplateParser();  
    public ITemplateWriter getTemplateWriter();  
}
```

Each *template mode handler* defines everything Thymeleaf needs to process templates in a specific mode: a *parser* (`ITemplateParser`) that is able to convert a template into a DOM tree, and a *writer* (`ITemplateWriter`) that is able to convert a DOM tree into the desired result format once it is processed.

Several template modes are provided by Thymeleaf out-of-the-box, defined at the `org.thymeleaf.templatemode.StandardTemplateModeHandlers` class, and they are preregistered for every `TemplateEngine` instance. Their names are:

- XML: for XML that does not require validation during parsing.
- VALIDXML: for XML that should be validated during parsing.
- XHTML: for XHTML 1.0 or 1.1 templates that do not need validation.
- VALIDXHTML: for XHTML 1.0 or 1.1 templates that should be validated during parsing.
- HTML5: for HTML5 templates that are well-formed XML documents.
- LEGACYHTML5: for HTML5 templates that are not well-formed XML documents, and therefore need a previous preprocessing step for tag balancing, syntax correction, etc.

For parsing templates in these modes, Thymeleaf offers a set of parser implementations that live at the `org.thymeleaf.templateparser` package. These parsers come in both SAX and DOM flavours –both validating and non-validating–, and there's also a `nekoHTML`-based HTML parser that allows parsing code that is not well-formed XML (for example, with unclosed tags).

By default, all standard modes use SAX parsing, except `LEGACYHTML5` which uses `nekoHTML`.

As for writers, two implementations of `ITemplateWriter` are provided: one for XHTML and HTML5, and another one for XML. Both live in the `org.thymeleaf.templatewriter` package.



## 4 CREATING OUR OWN DIALECT


### 4.1 EXTRATHYME: A WEBSITE FOR THYMELAND'S FOOTBALL LEAGUE

Football is a popular sport in Thymeland<sup>1</sup>. 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.

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



Thymeland's football website

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 Trogodytes (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

<sup>1</sup> 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>
      
    </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 looooooser</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.)

## 4.2 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 towards 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("classforposition");
    }

    public int getPrecedence() {
        return 12000;
    }

    @Override
    protected Map<String, String> getModifiedAttributeValues(
        final Arguments arguments, final Element element, final String attributeName) {

        /*
         * Obtain the attribute value
         */
        final String attributeValue = element.getAttributeValue(attributeName);

        /*
         * Process (parse and execute) the attribute value, specified as a
         * Thymeleaf Standard Expression.
         */
        final Integer position =
            (Integer) StandardExpressionProcessor.processExpression(arguments, attributeValue);

        /*
         * 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 Element element, final String attributeName, 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 Element element, final String attributeName, final String newAttributeName) {
        // If the resulting 'class' attribute is empty, do not show it at all.
        return true;
    }

    @Override
    protected boolean recomputeProcessorsAfterExecution(final Arguments arguments,
        final Element element, final String attributeName) {
        // There is no need to recompute the element after this processor has executed
        return false;
    }
}

```

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 expressions (also called *Thymeleaf Standard Expressions*) we make use of the `StandardExpressionProcessor` class methods:

```

final Integer position =
    (Integer) StandardExpressionProcessor.processExpression(arguments, attributeValue);

```

## 4.3 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.

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("remarkforposition");
}

public int getPrecedence() {
    return 12000;
}

@Override
protected String getText(final Arguments arguments, final Element element, final String attributeName) {
    /*
     * Obtain the attribute value
     */
    final String attributeValue = element.getAttributeValue(attributeName);

    /*
     * Process (parse and execute) the attribute value, specified as a
     * Thymeleaf Standard Expression.
     */
    final Integer position =
        (Integer) StandardExpressionProcessor.processExpression(arguments, attributeValue);

    /*
     * 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 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, "remarks." + remark.toString(), new Object[0]);
}
}

```

Note that we are accessing the message externalization system with:

```

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

```

But this in fact is not the only way. As previously mentioned in this guide, the `AbstractProcessor` 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' processors.

## 4.4 AN ELEMENT PROCESSOR FOR OUR HEADLINES

The third and last processor we will have to write is an element (tag) processor. As their name implies, element processors are triggered by element names instead of attribute names, and they have one advantage and also one disadvantage with respect to attribute 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 (something called *natural templating*)

This processor will extend `org.thymeleaf.processor.element.AbstractElementProcessor`, but as we did with our attribute processors, instead of extending it directly, we will use a more specialized abstract convenience class as a base for our processor class: `AbstractMarkupSubstitutionElementProcessor`. This is a base element processor that simply expects you to generate the DOM nodes that will substitute the host tag when the template is processed.

And this is our code:

```
public class HeadlinesElementProcessor extends AbstractMarkupSubstitutionElementProcessor {  
    private final Random rand = new Random(System.currentTimeMillis());  
    public HeadlinesTagProcessor() {  
        super("headlines");  
    }  
}
```

```

    }

    public int getPrecedence() {
        return 1000;
    }

    @Override
    protected List<Node> getMarkupSubstitutes(final Arguments arguments, 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.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 Element container = new Element("div");
        container.setAttribute("class", "headlines");

        final Text text = new Text(headlineText);
        container.addChild(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.getAttributeValue("order");
```

## 4.5 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 valuetag/attribute processor
     * associated with it, an exception is thrown.
     */
    public boolean isLenient() {
        return false;
    }

    /**
     * Two attribute processors are declared: 'classforposition' and
     * 'remarkforposition'. Also one element processor: the 'headlines'
     * tag.
     */
    @Override
    public Set<IProcessor> getProcessors() {
        final Set<IProcessor> processors = new HashSet<IProcessor>();
        processors.add(new ClassForPositionAttrProcessor());
        processors.add(new RemarkForPositionAttrProcessor());
        processors.add(new HeadlinesElementProcessor());
        return processors;
    }

}
```

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.