TUTORIAL: EXTENDING THYMELEAF

June 26th, 2011. THIS TUTORIAL IS WORK IN PROGRESS

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

CONTENTS

1 DIALECTS

If you've read the “Using Thymeleaf” tutorial before getting here (which you should have), you should know that what you've learned is not exactly “Thymeleaf”, but rather the “Thymeleaf Standard Dialect”.

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 the Standard Dialect is, obviously, “th”. Prefix can be null so that you can define attribute/tag processors for non-namespaced tags (for example, standard <p>, 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.

The Standard Dialect is 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 thyme and are used by them.

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, Thyme 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 to resolve DTDs linked from your templates.

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 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.
- The value processor dependencies. An attribute processor can make use of several value processors, and this method allows Thymeleaf to validate your dialect on startup making sure all the required value processors are declared.

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.

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 this method allows Thymeleaf to validate your dialect on startup making sure all the required value processors are declared.

Equivalently to attribute processors, the “process(...)” method receives data such as configuration, context, resolved template data, and the DOM element.

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, in turn delegating 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 Stirng 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 Thyme 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 for executing, and so the parameters passed to its methods will vary consequently.

5 HELLOWORLDDIALECT

Let's create a very simple non-validating dialect including only a “hello:world” attribute that will output a greeting for the world that is passed as attribute value, like:

```
    <p hello:world=”Earth”>Hello, World!</p>
```

Which should be processed into:

```
    <p>Hello, Earth!</p>
```

5.1 THE ATTRIBUTE PROCESSOR

Let's have a look at the implementation we need for our attribute processor:

```
public class WorldAttrProcessor extends AbstractAttrProcessor {

    public WorldAttrProcessor() {
        super();
    }

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

    public Integer getPrecedence() {
        return Integer.valueOf(100);
    }

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

        final String world = attribute.getValue();
        element.setTextContent("Hello, " + world + "!");

        // This means that the “hello:world” attribute should be removed
        // from the DOM once this processor has been executed.
        return AttrProcessResult.forRemoveAttribute();

    }

}
```

Pretty simple: we specify the attribute name, the precedence (which can be any value because it is our only attribute), and a process method which modifies the DOM Element in the way we want.

5.2 The Dialect

Now we need an IDialect implementation including our new attribute processor. Most used-defined dialects would extend the StandardDialect, and add/remove features to it, but our HelloWorldDialect is so simple that we will just extend AbstractXHTMLEnabledDialect:

```
public class HelloWorldDialect extends AbstractXHTMLEnabledDialect {

    public HelloWorldDialect() {
        super();
    }

    public String getPrefix() {
        return "hello";
    }

    public boolean isLenient() {
        return false;
    }

    @Override
    public Set<IAttrProcessor> getAttrProcessors() {
        final Set<IAttrProcessor> attrProcessors = new HashSet<IAttrProcessor>();
        attrProcessors.add(new WorldAttrProcessor());
        return attrProcessors;
    }

}
```

Simple as well. We just declare the prefix and the attribute processor we created. And that's all.

Let's use our brand new dialect now.

5.3 Using it

We can now use our HelloWorldDialect very easily:

```
public String processTemplate(final String templateName) {

    final ITemplateResolver templateResolver = new ClassLoaderTemplateResolver();
    templateResolver.setTemplateMode(TemplateMode.XHTML);

    final IDialect dialect = new HelloWorldDialect();

    final TemplateEngine templateEngine = new TemplateEngine();
    templateEngine.setDialect(dialect);
    templateEngine.setTemplateResolver(templateResolver);

    final IContext ctx = new Context(new Locale("es", "ES"));

    return templateEngine.process(templateName, context);

}
```

Then we create our template:

```
<!DOCTYPE html PUBLIC
          "-//W3C//DTD XHTML 1.0 Strict//EN"
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:hello="http://helloworld">

  <head>
    <title>Hello World Test</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  </head>

  <body id="body">
    <p hello:world="Earth">Hello, World!</p>
  </body>

</html>
```

And process it:

```
<!DOCTYPE html PUBLIC
          "-//W3C//DTD XHTML 1.0 Strict//EN"
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">

  <head>
    <title>Hello World Test</title>
    <meta content="text/html; charset=UTF-8" http-equiv="Content-Type"/>
  </head>

  <body id="body">
    <p>Hello, Earth!</p>
  </body>

</html>
```