Tutorial: Using Thymeleaf

Document version: 20141222 - 22 December 2014
Project version: 2.1.4.RELEASE

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

1 Introducing Thymeleaf

1.1 What is Thymeleaf?

Thymeleaf is a Java library. It is an XML/XHTML/HTML5 template engine able to apply a set of transformations to template files in order to display data and/or text produced by your applications.

It is better suited for serving XHTML/HTML5 in web applications, but it can process any XML file, be it in web or in standalone applications.

The main goal of Thymeleaf is to provide an elegant and well-formed way of creating templates. In order to achieve this, it is based on XML tags and attributes that define the execution of predefined logic on the *DOM (Document Object Model)*, instead of explicitly writing that logic as code inside the template.

Its architecture allows a fast processing of templates, relying on intelligent caching of parsed files in order to use the least possible amount of I/O operations during execution.

And last but not least, Thymeleaf has been designed from the beginning with XML and Web standards in mind, allowing you to create fully validating templates if that is a need for you.

1.2 What kind of templates can Thymeleaf process?

Out-of-the-box, Thymeleaf allows you to process six kinds of templates, each of which is called a Template Mode:

- XML
- Valid XML
- XHTML
- Valid XHTML
- HTML5
- Legacy HTML5

All of these modes refer to well-formed XML files except the *Legacy HTML5* mode, which allows you to process HTML5 files with features such as standalone (not closed) tags, tag attributes without a value or not written between quotes. In order to process files in this specific mode, Thymeleaf will first perform a transformation that will convert your files to well-formed XML files which are still perfectly valid HTML5 (and are in fact the recommended way to create HTML5 code)[1].

Also note that validation is only available for XML and XHTML templates.

Nevertheless, these are not the only types of template that Thymeleaf can process, and the user is always able to define his/her own mode by specifying both a way to *parse* templates in this mode and a way to *write* the results. This way, anything that can be modelled as a DOM tree (be it XML or not) could effectively be processed as a template by Thymeleaf.

1.3 Dialects: The Standard Dialect

Thymeleaf is an extremely extensible template engine (in fact it should be better called a *template engine framework*) that allows you to completely define the DOM nodes that will be processed in your templates and also how they will be processed.

An object that applies some logic to a DOM node is called a *processor*, and a set of these processors —plus some extra artifacts— is called a dialect, of which Thymeleaf's core library provides one out-of-the-box called the *Standard Dialect*, which should be enough for the needs of a big percent of users.

The Standard Dialect is the dialect this tutorial covers. Every attribute and syntax feature you will learn about in the following pages is defined by this dialect, even if that isn't explicitly mentioned.

Of course, users may create their own dialects (even extending the Standard one) if they want to define their own processing logic while taking advantage of the library's advanced features. A Template Engine can be configured several dialects at a time.

> The official thymeleaf-spring3 and thymeleaf-spring4 integration packages both define a dialect called the "SpringStandard Dialect", mostly equivalent to the Standard Dialect but with small adaptations to make better use of some features in Spring Framework (for example, by using Spring Expression Language instead of Thymeleaf's standard OGNL). So if you are a Spring MVC user you are not wasting your time, as almost everything you learn here will be of use in your Spring applications.

The Thymeleaf Standard Dialect can process templates in any mode, but is especially suited for web-oriented template modes (XHTML and HTML5 ones). Besides HTML5, it specifically supports and validates the following XHTML specifications: *XHTML 1.0 Transitional*, *XHTML 1.0 Strict*, *XHTML 1.0 Frameset*, and *XHTML 1.1*.

Most of the processors of the Standard Dialect are *attribute processors*. This allows browsers to correctly display XHTML/HTML5 template files even before being processed, because they will simply ignore the additional attributes. For example, while a JSP using tag libraries could include a fragment of code not directly displayable by a browser like:

```
<form:inputText name="userName" value="${user.name}" />
```

...the Thymeleaf Standard Dialect would allow us to achieve the same functionality with:

```
<input type="text" name="userName" value="James Carrot" th:value="${user.name}" />
```

Which not only will be correctly displayed by browsers, but also allow us to (optionally) specify a value attribute in it ("James Carrot", in this case) that will be displayed when the prototype is statically opened in a browser, and that will be substituted by the value resulting from the evaluation of `${user.name}` during Thymeleaf processing of the template.

If needed, this will allow your designer and developer to work on the very same template file and reduce the effort required to transform a static prototype into a working template file. The ability to do this is a feature usually called *Natural Templating*.

1.4 Overall Architecture

Thymeleaf's core is a DOM processing engine. Specifically, it uses its own high-performance DOM implementation —not the standard DOM API— for building in-memory tree representations of your templates, on which it later operates by traversing their nodes and executing processors on them that modify the DOM according to the current *configuration* and the set of data that is passed to the template for its representation —known as the context.

The use of a DOM template representation makes it very well suited for web applications because web documents are very often represented as object trees (in fact DOM trees are the way browsers represent web pages in memory). Also, building on the idea that most web applications use only a few dozen templates, that these are not big files and that they don't normally change while the application is running, Thymeleaf's usage of an in-memory cache of parsed template DOM trees allows it to be fast in production environments, because very little I/O is needed (if any) for most template processing operations.

> If you want more detail, later in this tutorial there is an entire chapter dedicated to caching and to the way Thymeleaf optimizes memory and resource usage for faster operation.

Nevertheless, there is a restriction: this architecture also requires the use of bigger amounts of memory space for

each template execution than other template parsing/processing approaches, which means that you should not use the library for creating big data XML documents (as opposed to web documents). As a general rule of thumb (and always depending on the memory size of your JVM), if you are generating XML files with sizes around the tens of megabytes in a single template execution, you probably should not be using Thymeleaf.

> The reason we consider this restriction only applies to data XML files and not web XHTML/HTML5 is that you should never generate web documents so big that your users' browsers set ablaze and/or explode – remember that these browsers will also have to create DOM trees for your pages!

1.5 Before going any further, you should read...

Thymeleaf is especially suited for working in web applications. And web applications are based on a series of standards that everyone should know very well but few do – even if they have been working with them for years.

With the advent of HTML5, the state of the art in web standards today is more confusing than ever... *are we going back from XHTML to HTML? Will we abandon XML syntax? Why is nobody talking about XHTML 2.0 anymore?*

So before going any further in this tutorial, you are strongly advised to read an article on Thymeleaf's web site called *"From HTML to HTML (via HTML)"*, which you can find at this address:
http://www.thymeleaf.org/fromhtmltohtmlviahtml.html

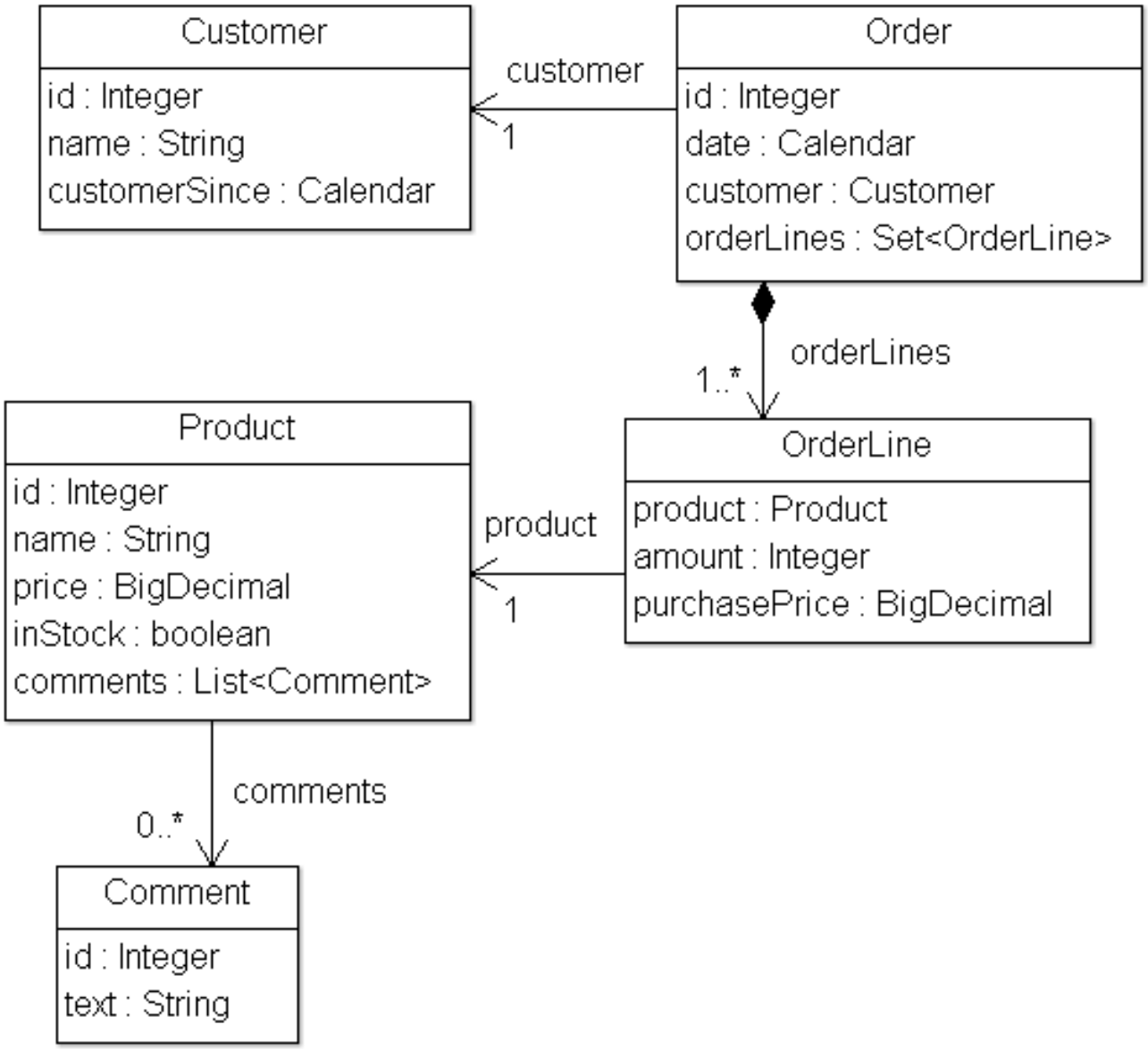
2 The Good Thymes Virtual Grocery

2.1 A website for a grocery

In order to better explain the concepts involved in processing templates with Thymeleaf, this tutorial will use a demo application you can download from the project web site.

This application represents the web site of an imaginary virtual grocery, and will provide us with the adequate scenarios to exemplify diverse Thymeleaf features.

We will need a quite simple set of model entities for our application: `Products` which are sold to `Customers` by creating `Orders`. We will also be managing `Comments` about those `Products`:



Example application model

Our small application will also have a very simple service layer, composed by `Service` objects containing methods like:

```
public class ProductService {
```

```
    ...

    public List<Product> findAll() {
        return ProductRepository.getInstance().findAll();
    }

    public Product findById(Integer id) {
        return ProductRepository.getInstance().findById(id);
    }

}
```

Finally, at the web layer our application will have a filter that will delegate execution to Thymeleaf-enabled commands depending on the request URL:

```
private boolean process(HttpServletRequest request, HttpServletResponse response)
        throws ServletException {

    try {

        /*
         * Query controller/URL mapping and obtain the controller
         * that will process the request. If no controller is available,
         * return false and let other filters/servlets process the request.
         */
        IGTVGController controller = GTVGApplication.resolveControllerForRequest(request);
        if (controller == null) {
            return false;
        }
        /*
         * Obtain the TemplateEngine instance.
         */
        TemplateEngine templateEngine = GTVGApplication.getTemplateEngine();

        /*
         * Write the response headers
         */
        response.setContentType("text/html;charset=UTF-8");
        response.setHeader("Pragma", "no-cache");
        response.setHeader("Cache-Control", "no-cache");
        response.setDateHeader("Expires", 0);

        /*
         * Execute the controller and process view template,
         * writing the results to the response writer.
         */
        controller.process(
                request, response, this.servletContext, templateEngine);

        return true;

    } catch (Exception e) {
        throw new ServletException(e);
    }

}
```

This is our `IGTVGController` interface:

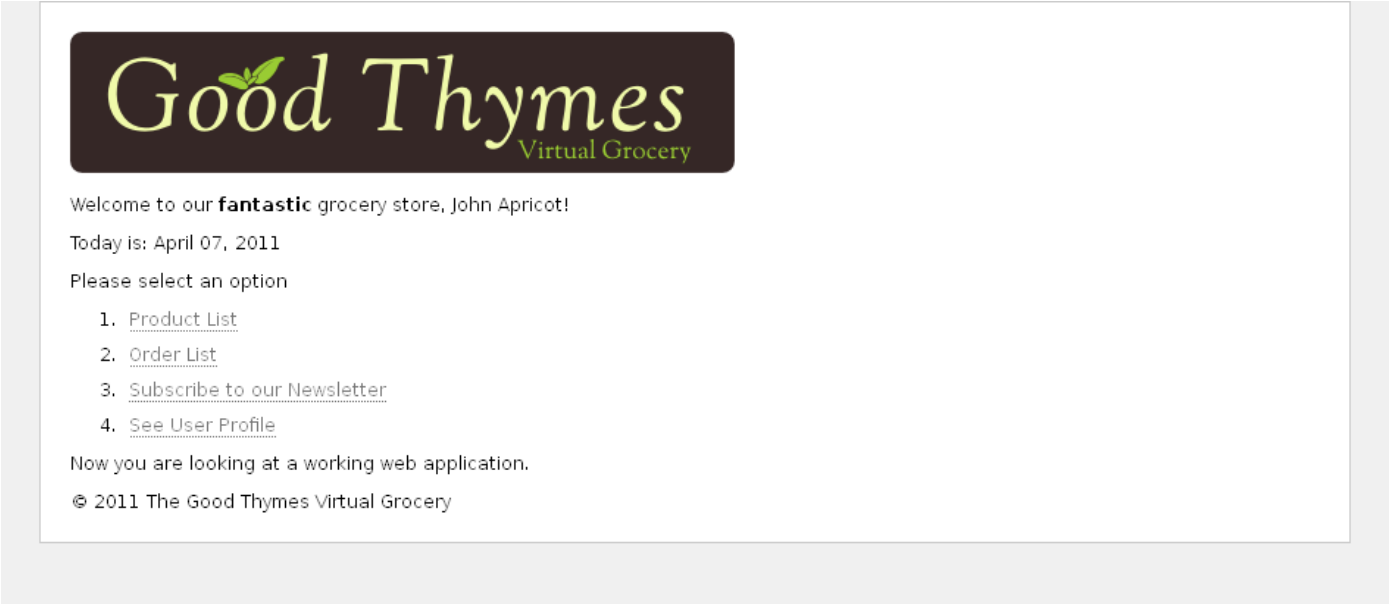
```
public interface IGTVGController {

    public void process(
            HttpServletRequest request, HttpServletResponse response,
            ServletContext servletContext, TemplateEngine templateEngine);

}
```

All we have to do now is create implementations of the `IGTVGController` interface, retrieving data from the services and processing templates using the `TemplateEngine` object.

In the end, it will look like this:



Example application home page

But first let's see how that template engine is initialized.

2.2 Creating and configuring the Template Engine

The *process(...)* method in our filter contained this sentence:

```
TemplateEngine templateEngine = GTVGApplication.getTemplateEngine();
```

Which means that the *GTVGApplication* class is in charge of creating and configuring one of the most important objects in a Thymeleaf-enabled application: The `TemplateEngine` instance.

Our `org.thymeleaf.TemplateEngine` object is initialized like this:

```
public class GTVGApplication {


    ...
    private static TemplateEngine templateEngine;
    ...


    static {
        ...
        initializeTemplateEngine();
        ...
    }


    private static void initializeTemplateEngine() {

        ServletContextTemplateResolver templateResolver =
            new ServletContextTemplateResolver();
        // XHTML is the default mode, but we set it anyway for better understanding of code
        templateResolver.setTemplateMode("XHTML");
        // This will convert "home" to "/WEB-INF/templates/home.html"
        templateResolver.setPrefix("/WEB-INF/templates/");
        templateResolver.setSuffix(".html");
        // Template cache TTL=1h. If not set, entries would be cached until expelled by LRU
        templateResolver.setCacheTTLMs(3600000L);

        templateEngine = new TemplateEngine();
        templateEngine.setTemplateResolver(templateResolver);

    }
```

```
    ...

}
```

Of course there are many ways of configuring a `TemplateEngine` object, but for now these few lines of code will teach us enough about the steps needed.

The Template Resolver

Let's start with the Template Resolver:

```
ServletContextTemplateResolver templateResolver = new ServletContextTemplateResolver();
```

Template Resolvers are objects that implement an interface from the Thymeleaf API called `org.thymeleaf.templateresolver.ITemplateResolver`:

```
public interface ITemplateResolver {

    ...

    /*
     * Templates are resolved by String name (templateProcessingParameters.getTemplateName())
     * Will return null if template cannot be handled by this template resolver.
     */
    public TemplateResolution resolveTemplate(
            TemplateProcessingParameters templateProcessingParameters);

}
```

These objects are in charge of determining how our templates will be accessed, and in this GTVG application, the `org.thymeleaf.templateresolver.ServletContextTemplateResolver` implementation that we are using specifies that we are going to retrieve our template files as resources from the *Servlet Context*: an application-wide `javax.servlet.ServletContext` object that exists in every Java web application, and that resolves resources considering the web application root as the root for resource paths.

But that's not all we can say about the template resolver, because we can set some configuration parameters on it. First, the template mode, one of the standard ones:

```
templateResolver.setTemplateMode("XHTML");
```

XHTML is the default template mode for `ServletContextTemplateResolver`, but it is good practice to establish it anyway so that our code documents clearly what is going on.

```
templateResolver.setPrefix("/WEB-INF/templates/");
templateResolver.setSuffix(".html");
```

These *prefix* and *suffix* do exactly what it looks like: modify the template names that we will be passing to the engine for obtaining the real resource names to be used.

Using this configuration, the template name *"product/list"* would correspond to:

```
servletContext.getResourceAsStream("/WEB-INF/templates/product/list.html")
```

Optionally, the amount of time that a parsed template living in cache will be considered valid can be configured at the Template Resolver by means of the *cacheTTLMs* property:

```
templateResolver.setCacheTTLMs(3600000L);
```

Of course, a template can be expelled from cache before that TTL is reached if the max cache size is reached and it is

the oldest entry currently cached.

> Cache behaviour and sizes can be defined by the user by implementing the `ICacheManager` interface or simply modifying the `StandardCacheManager` object set to manage caches by default.

We will learn more about template resolvers later. Now let's have a look at the creation of our Template Engine object.

The Template Engine

Template Engine objects are of class *org.thymeleaf.TemplateEngine*, and these are the lines that created our engine in the current example:

```
templateEngine = new TemplateEngine();
templateEngine.setTemplateResolver(templateResolver);
```

Rather simple, isn't it? All we need is to create an instance and set the Template Resolver to it.

A template resolver is the only required parameter a `TemplateEngine` needs, although of course there are many others that will be covered later (message resolvers, cache sizes, etc). For now, this is all we need.

Our Template Engine is now ready and we can start creating our pages using Thymeleaf.

3 Using Texts

3.1 A multi-language welcome

Our first task will be to create a home page for our grocery site.

The first version we will write of this page will be extremely simple: just a title and a welcome message. This is our `/WEB-INF/templates/home.html` file:

```
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-4.dtd">

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

  <head>
    <title>Good Thymes Virtual Grocery</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <link rel="stylesheet" type="text/css" media="all"
          href="../../css/gtvg.css" th:href="@{/css/gtvg.css}" />
  </head>

  <body>

    <p th:text="#{home.welcome}">Welcome to our grocery store!</p>

  </body>

</html>
```

The first thing you will notice here is that this file is XHTML that can be correctly displayed by any browser, because it does not include any non-XHTML tags (and browsers ignore all attributes they don't understand, like `th:text`). Also, browsers will display it in standards mode (not in quirks mode), because it has a well-formed `DOCTYPE` declaration.

Next, this is also *valid* XHTML[2], because we have specified a Thymeleaf DTD which defines attributes like `th:text` so that your templates can be considered valid. And even more: once the template is processed (and all `th:*` attributes are removed), Thymeleaf will automatically substitute that DTD declaration in the `DOCTYPE` clause by a standard `XHTML 1.0 Strict` one (we will leave this DTD translation features for a later chapter).

A thymeleaf namespace is also being declared for `th:*` attributes:

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

Note that, if we hadn't cared about our template's validity or well-formedness at all, we could have simply specified a standard `XHTML 1.0 Strict DOCTYPE`, along with no xmlns namespace declarations:

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

<html>

  <head>
    <title>Good Thymes Virtual Grocery</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <link rel="stylesheet" type="text/css" media="all"
          href="../../css/gtvg.css" th:href="@{/css/gtvg.css}" />
  </head>

  <body>

    <p th:text="#{home.welcome}">Welcome to our grocery store!</p>

  </body>
```

```
</html>
```

...and this would still be perfectly processable by Thymeleaf in the XHTML mode (although probably our IDE would make our life quite miserable showing warnings everywhere).

But enough about validation. Now for the really interesting part of the template: let's see what that `th:text` attribute is about.

Using th:text and externalizing text

Externalizing text is extracting fragments of template code out of template files so that they can be kept in specific separate files (typically `.properties` files) and that they can be easily substituted by equivalent texts written in other languages (a process called internationalization or simply *i18n*). Externalized fragments of text are usually called "messages".

Messages have always a key that identifies them, and Thymeleaf allows you to specify that a text should correspond to a specific message with the `#{...}` syntax:

```
<p th:text="#{home.welcome}">Welcome to our grocery store!</p>
```

What we can see here are in fact two different features of the Thymeleaf Standard Dialect:

- The `th:text` attribute, which evaluates its value expression and sets the result of this evaluation as the body of the tag it is in, effectively substituting that "Welcome to our grocery store!" text we see in the code.
- The `#{home.welcome}` expression, specified in the *Standard Expression Syntax*, specifying that the text to be used by the `th:text` attribute should be the message with the `home.welcome` key corresponding to whichever locale we are processing the template with.

Now, where is this externalized text?

The location of externalized text in Thymeleaf is fully configurable, and it will depend on the specific `org.thymeleaf.messageresolver.IMessageResolver` implementation being used. Normally, an implementation based on `.properties` files will be used, but we could create our own implementations if we wanted, for example, to obtain messages from a database.

However, we have not specified a message resolver to our Template Engine during initialization, and that means that our application is using the *Standard Message Resolver*, implemented by class `org.thymeleaf.messageresolver.StandardMessageResolver`.

This standard message resolver expects to find messages for `/WEB-INF/templates/home.html` in .properties files in the same folder and with the same name as the template, like:

- `/WEB-INF/templates/home_en.properties` for English texts.
- `/WEB-INF/templates/home_es.properties` for Spanish language texts.
- `/WEB-INF/templates/home_pt_BR.properties` for Portuguese (Brazil) language texts.
- `/WEB-INF/templates/home.properties` for default texts (if locale is not matched).

Let's have a look at our `home_es.properties` file:

```
home.welcome=¡Bienvenido a nuestra tienda de comestibles!
```

This is all we need for making Thymeleaf process our template. Let's create our Home controller then.

Contexts

In order to process our template, we will create a `HomeController` class implementing the `IGTVGController` interface we saw before:

```
public class HomeController implements IGTVGController {

    public void process(
            HttpServletRequest request, HttpServletResponse response,
            ServletContext servletContext, TemplateEngine templateEngine) {

        WebContext ctx =
            new WebContext(request, response, servletContext, request.getLocale());
        templateEngine.process("home", ctx, response.getWriter());

    }

}
```

The first thing we can see here is the creation of a context. A Thymeleaf context is an object implementing the `org.thymeleaf.context.IContext` interface. Contexts should contain all the data required for an execution of the Template Engine in a variables map, and also reference the Locale that must be used for externalized messages.

```
public interface IContext {

    public VariablesMap<String,Object> getVariables();
    public Locale getLocale();
    ...

}
```

There is a specialized extension of this interface, `org.thymeleaf.context.IWebContext`:

```
public interface IWebContext extends IContext {

    public HttpSerlvetRequest getHttpServletRequest();
    public HttpSession getHttpSession();
    public ServletContext getServletContext();

    public VariablesMap<String,String[]> getRequestParameters();
    public VariablesMap<String,Object> getRequestAttributes();
    public VariablesMap<String,Object> getSessionAttributes();
    public VariablesMap<String,Object> getApplicationAttributes();

}
```

The Thymeleaf core library offers an implementation of each of these interfaces:

- `org.thymeleaf.context.Context` implements `IContext`
- `org.thymeleaf.context.WebContext` implements `IWebContext`

And as you can see in the controller code, `WebContext` is the one we will use. In fact we have to, because the use of a `ServletContextTemplateResolver` requires that we use a context implementing `IWebContext`.

```
WebContext ctx = new WebContext(request, servletContext, request.getLocale());
```

Only two of those three constructor arguments are required, because the default locale for the system will be used if none is specified (although you should never let this happen in real applications).

From the interface definition we can tell that `WebContext` will offer specialized methods for obtaining the request parameters and request, session and application attributes . But in fact `WebContext` will do a little bit more than just that:

- Add all the request attributes to the context variables map.
- Add a context variable called `param` containing all the request parameters.
- Add a context variable called `session` containing all the session attributes.
- Add a context variable called `application` containing all the ServletContext attributes.

Just before execution, a special variable is set into all context objects (implementations of `IContext`), including both

`Context` and `WebContext`, called the execution info (`execInfo`). This variable contains two pieces of data that can be used from within your templates:

- The template name (`${execInfo.templateName}`), the name specified for engine execution, and corresponding to the template being executed.
- The current date and time (`${execInfo.now}`), a `Calendar` object corresponding to the moment the template engine started its execution for this template.

Executing the template engine

With our context object ready, all we need is executing the template engine specifying the template name and the context, and passing on the response writer so that the response can be written to it:

```
templateEngine.process("home", ctx, response.getWriter());
```

Let's see the results of this using the Spanish locale:

```
<!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>Good Thymes Virtual Grocery</title>
    <meta content="text/html; charset=UTF-8" http-equiv="Content-Type"/>
    <link rel="stylesheet" type="text/css" media="all" href="/gtvg/css/gtvg.css" />
  </head>

  <body>

    <p>¡Bienvenido a nuestra tienda de comestibles!</p>

  </body>

</html>
```

3.2 More on texts and variables

Unescaped Text

The simplest version of our Home page seems to be ready now, but there is something we have not thought about... what if we had a message like this?

```
home.welcome=Welcome to our <b>fantastic</b> grocery store!
```

If we execute this template like before, we will obtain:

```
<p>Welcome to our &lt;b&gt;fantastic&lt;/b&gt; grocery store!</p>
```

Which is not exactly what we expected, because our `<b>` tag has been escaped and therefore it will be displayed at the browser.

This is the default behaviour of the th:text attribute. If we want Thymeleaf to respect our XHTML tags and not escape them, we will have to use a different attribute: `th:utext` (for "unescaped text"):

```
<p th:utext="#{home.welcome}">Welcome to our grocery store!</p>
This will output our message just like we wanted it:
<p>Welcome to our <b>fantastic</b> grocery store!</p>
```

Using and displaying variables

Now let's add some more contents to our home page. For example, we could want to display the date below our welcome message, like this:

```
Welcome to our fantastic grocery store!

Today is: 12 july 2010
```

First of all, we will have to modify our controller so that we add that date as a context variable:

```
public void process(
        HttpServletRequest request, HttpServletResponse response,
        ServletContext servletContext, TemplateEngine templateEngine) {

    SimpleDateFormat dateFormat = new SimpleDateFormat("dd MMMM yyyy");
    Calendar cal = Calendar.getInstance();

    WebContext ctx =
        new WebContext(request, response, servletContext, request.getLocale());
    ctx.setVariable("today", dateFormat.format(cal.getTime()));

    templateEngine.process("home", ctx, response.getWriter());

}
```

We have added a `String` today variable to our context, and now we can display it in our template:

```
<body>

  <p th:utext="#{home.welcome}">Welcome to our grocery store!</p>

  <p>Today is: <span th:text="${today}">13 February 2011</span></p>

</body>
```

As you can see, we are still using the `th:text` attribute for the job (and that's correct, because we want to substitute the tag's body), but the syntax is a little bit different this time and instead of a `#{...}` expression value, we are using a `${...}` one. This is a variable expression value, and it contains an expression in a language called *OGNL (Object-Graph Navigation Language)* that will be executed on the context variables map.

The `${today}` expression simply means "get the variable called today", but these expressions could be more complex (like `${user.name}` for "get the variable called user, and call its `getName()` method").

There are quite a lot of possibilities in attribute values: messages, variable expressions... and quite a lot more. Next chapter will show us what all these possibilities are.

4 Standard Expression Syntax

We will make a small break in the development of our grocery virtual store to learn about one of the most important parts of the Thymeleaf Standard Dialect: the Thymeleaf Standard Expression syntax.

We have already seen two types of valid attribute values expressed in this syntax: message and variable expressions:

```
<p th:utext="#{home.welcome}">Welcome to our grocery store!</p>

<p>Today is: <span th:text="${today}">13 february 2011</span></p>
```

But there are more types of value we don't know yet, and more interesting detail to know about the ones we already know. First, let's see a quick summary of the Standard Expression features:

- Simple expressions:
  - Variable Expressions: `${...}`
  - Selection Variable Expressions: `*{...}`
  - Message Expressions: `#{...}`
  - Link URL Expressions: `@{...}`
- Literals
  - Text literals: `'one text'`, `'Another one!'`,...
  - Number literals: `0`, `34`, `3.0`, `12.3`,...
  - Boolean literals: `true`, `false`
  - Null literal: `null`
  - Literal tokens: `one`, `sometext`, `main`,...
- Text operations:
  - String concatenation: `+`
  - Literal substitutions: `|The name is ${name}|`
- Arithmetic operations:
  - Binary operators: `+`, `-`, `*`, `/`, `%`
  - Minus sign (unary operator): `-`
- Boolean operations:
  - Binary operators: `and`, `or`
  - Boolean negation (unary operator): `!`, `not`
- Comparisons and equality:
  - Comparators: `>`, `<`, `>=`, `<=` (`gt`, `lt`, `ge`, `le`)
  - Equality operators: `==`, `!=` (`eq`, `ne`)
- Conditional operators:
  - If-then: `(if) ? (then)`
  - If-then-else: `(if) ? (then) : (else)`
  - Default: `(value) ?: (defaultvalue)`

All these features can be combined and nested:

```
'User is of type ' + (${user.isAdmin()} ? 'Administrator' : (${user.type} ?: 'Unknown'))
```

4.1 Messages

As we already know, `#{...}` message expressions allow us to link this:

```
<p th:utext="#{home.welcome}">Welcome to our grocery store!</p>
```

...to this:

```
home.welcome=¡Bienvenido a nuestra tienda de comestibles!
```

But there's one aspect we still haven't thought of: what happens if the message text is not completely static? What if, for example, our application knew who is the user visiting the site at any moment and we wanted to greet him/her by name?

```
<p>¡Bienvenido a nuestra tienda de comestibles, John Apricot!</p>
```

This means we would need to add a parameter to our message. Just like this:

```
home.welcome=¡Bienvenido a nuestra tienda de comestibles, {0}!
```

Parameters are specified according to the `java.text.MessageFormat` standard syntax, which means you could add format to numbers and dates as specified in the API docs for that class.

In order to specify a value for our parameter, and given an HTTP session attribute called `user`, we would have:

```
<p th:utext="#{home.welcome(${session.user.name})}">
  Welcome to our grocery store, Sebastian Pepper!
</p>
```

If needed, several parameters could be specified, separated by commas. In fact, the message key itself could come from a variable:

```
<p th:utext="#{${welcomeMsgKey}(${session.user.name})}">
  Welcome to our grocery store, Sebastian Pepper!
</p>
```

4.2 Variables

We already mentioned that `${...}` expressions are in fact OGNL (Object-Graph Navigation Language) expressions executed on the map of variables contained in the context.

> For detailed info about OGNL syntax and features, you should read the OGNL Language Guide at:
> http://commons.apache.org/ognl/

From OGNL's syntax, we know that this:

```
<p>Today is: <span th:text="${today}">13 february 2011</span>.</p>
```

...is in fact equivalent to this:

```
ctx.getVariables().get("today");
```

But OGNL allows us to create quite more powerful expressions, and that's how this:

```
<p th:utext="#{home.welcome(${session.user.name})}">
  Welcome to our grocery store, Sebastian Pepper!
</p>
```

...does in fact obtain the user name by executing:

```
((User) ctx.getVariables().get("session").get("user")).getName();
```

But getter method navigation is just one of OGNL's features. Let's see some more:

```
/*
 * Access to properties using the point (.). Equivalent to calling property getters.
 */
${person.father.name}

/*
 * Access to properties can also be made by using brackets ([]) and writing
 * the name of the property as a variable or between single quotes.
 */
${person['father']['name']}

/*
 * If the object is a map, both dot and bracket syntax will be equivalent to
 * executing a call on its get(...) method.
 */
${countriesByCode.ES}
${personsByName['Stephen Zucchini'].age}

/*
 * Indexed access to arrays or collections is also performed with brackets,
 * writing the index without quotes.
 */
${personsArray[0].name}

/*
 * Methods can be called, even with arguments.
 */
${person.createCompleteName()}
${person.createCompleteNameWithSeparator('-')}
```

Expression Basic Objects

When evaluating OGNL expressions on the context variables, some objects are made available to expressions for higher flexibility. These objects will be referenced (per OGNL standard) starting with the `#` symbol:

- `#ctx`: the context object.
- `#vars:` the context variables.
- `#locale`: the context locale.
- `#httpServletRequest`: (only in Web Contexts) the `HttpServletRequest` object.
- `#httpSession`: (only in Web Contexts) the `HttpSession` object.

So we can do this:

```
Established locale country: <span th:text="${#locale.country}">US</span>.
```

You can read the full reference of these objects in the Appendix A.

Expression Utility Objects

Besides these basic objects, Thymeleaf will offer us a set of utility objects that will help us perform common tasks in our expressions.

- `#dates`: utility methods for `java.util.Date` objects: formatting, component extraction, etc.
- `#calendars`: analogous to `#dates`, but for `java.util.Calendar` objects.
- `#numbers`: utility methods for formatting numeric objects.
- `#strings`: utility methods for `String` objects: contains, startsWith, prepending/appending, etc.
- `#objects`: utility methods for objects in general.
- `#bools`: utility methods for boolean evaluation.
- `#arrays`: utility methods for arrays.
- `#lists`: utility methods for lists.
- `#sets`: utility methods for sets.

- `#maps`: utility methods for maps.
- `#aggregates`: utility methods for creating aggregates on arrays or collections.
- `#messages`: utility methods for obtaining externalized messages inside variables expressions, in the same way as they would be obtained using #{...} syntax.
- `#ids`: utility methods for dealing with id attributes that might be repeated (for example, as a result of an iteration).

You can check what functions are offered by each of these utility objects in the Appendix B.

Reformatting dates in our home page

Now we know about these utility objects, we could use them to change the way in which we show the date in our home page. Instead of doing this in our `HomeController`:

```
SimpleDateFormat dateFormat = new SimpleDateFormat("dd MMMM yyyy");
Calendar cal = Calendar.getInstance();

WebContext ctx = new WebContext(request, servletContext, request.getLocale());
ctx.setVariable("today", dateFormat.format(cal.getTime()));

templateEngine.process("home", ctx, response.getWriter());
```

...we can do just this:

```
WebContext ctx = new WebContext(request, servletContext, request.getLocale());
ctx.setVariable("today", Calendar.getInstance());

templateEngine.process("home", ctx, response.getWriter());
```

...and then perform date formatting in the view layer itself:

```
<p>
  Today is: <span th:text="${#calendars.format(today,'dd MMMM yyyy')}">13 May 2011</span>
</p>
```

4.3 Expressions on selections (asterisk syntax)

Variable expressions not only can be written in `${...}` expressions, but also in `*{...}` ones.

There is an important difference, though: the asterisk syntax evaluates expressions on selected objects rather than on the whole context variables map. This is: as long as there is no selected object, the dollar and the asterisk syntaxes do exactly the same.

And what is that object selection thing? A `th:object` attribute. Let's use it in our user profile (`userprofile.html`) page:

```
  <div th:object="${session.user}">
    <p>Name: <span th:text="*{firstName}">Sebastian</span>.</p>
    <p>Surname: <span th:text="*{lastName}">Pepper</span>.</p>
    <p>Nationality: <span th:text="*{nationality}">Saturn</span>.</p>
  </div>
```

Which is exactly equivalent to:

```
<div>
  <p>Name: <span th:text="${session.user.firstName}">Sebastian</span>.</p>
  <p>Surname: <span th:text="${session.user.lastName}">Pepper</span>.</p>
  <p>Nationality: <span th:text="${session.user.nationality}">Saturn</span>.</p>
</div>
```

Of course, dollar and asterisk syntax can be mixed:

```
<div th:object="${session.user}">
  <p>Name: <span th:text="*{firstName}">Sebastian</span>.</p>
  <p>Surname: <span th:text="${session.user.lastName}">Pepper</span>.</p>
  <p>Nationality: <span th:text="*{nationality}">Saturn</span>.</p>
</div>
```

When an object selection is in place, the selected object will be also available to dollar expressions as the `#object` expression variable:

```
<div th:object="${session.user}">
  <p>Name: <span th:text="${#object.firstName}">Sebastian</span>.</p>
  <p>Surname: <span th:text="${session.user.lastName}">Pepper</span>.</p>
  <p>Nationality: <span th:text="*{nationality}">Saturn</span>.</p>
</div>
```

As said, if no object selection has been performed, dollar and asterisk syntaxes are exactly equivalent.

```
<div>
  <p>Name: <span th:text="*{session.user.name}">Sebastian</span>.</p>
  <p>Surname: <span th:text="*{session.user.surname}">Pepper</span>.</p>
  <p>Nationality: <span th:text="*{session.user.nationality}">Saturn</span>.</p>
</div>
```

4.4 Link URLs

Because of their importance, URLs are first-class citizens in web application templates, and the *Thymeleaf Standard Dialect* has a special syntax for them, the `@` syntax: `@{...}`

There are different types of URLs:

- Absolute URLs, like `http://www.thymeleaf.org`
- Relative URLs, which can be:
  - Page-relative, like `user/login.html`
  - Context-relative, like `/itemdetails?id=3` (context name in server will be added automatically)
  - Server-relative, like `~/billing/processInvoice` (allows calling URLs in another context (= application) in the same server.
  - Protocol-relative URLs, like `//code.jquery.com/jquery-2.0.3.min.js`

Thymeleaf can handle absolute URLs in any situation, but for relative ones it will require you to use a context object that implements the `IWebContext` interface, which contains some info coming from the HTTP request and needed to create relative links.

Let's use this new syntax. Meet the `th:href` attribute:

```
<!-- Will produce 'http://localhost:8080/gtvg/order/details?orderId=3' (plus rewriting) -->
<a href="details.html"
   th:href="@{http://localhost:8080/gtvg/order/details(orderId=${o.id})}">view</a>

<!-- Will produce '/gtvg/order/details?orderId=3' (plus rewriting) -->
<a href="details.html" th:href="@{/order/details(orderId=${o.id})}">view</a>

<!-- Will produce '/gtvg/order/3/details' (plus rewriting) -->
<a href="details.html" th:href="@{/order/{orderId}/details(orderId=${o.id})}">view</a>
```

Some things to note here:

- `th:href` is an attribute modifier attribute: once processed, it will compute the link URL to be used and set the href attribute of the `<a>` tag to this URL.

- We are allowed to use expressions for URL parameters (as you can see in `orderId=${o.id}`). The required URL-encoding operations will also be automatically performed.
- If several parameters are needed, these will be separated by commas like `@{/order/process(execId=${execId},execType='FAST')}`
- Variable templates are also allowed in URL paths, like `@{/order/{orderId}/details(orderId=${orderId})}`
- Relative URLs starting with `/` (like `/order/details`) will be automatically prefixed the application context name.
- If cookies are not enabled or this is not yet known, a `";jsessionid=..."` suffix might be added to relative URLs so that session is preserved. This is called *URL Rewriting*, and Thymeleaf allows you to plug in your own rewriting filters by using the `response.encodeURL(...)` mechanism from the Servlet API for every URL.
- The `th:href` tag allowed us to (optionally) have a working static `href` attribute in our template, so that our template links remained navigable by a browser when opened directly for prototyping purposes.

As was the case with the message syntax (`#{...}`), URL bases can also be the result of evaluating another expression:

```
<a th:href="@{${url}(orderId=${o.id})}">view</a>
<a th:href="@{'/details/'+${user.login}(orderId=${o.id})}">view</a>
```

A menu for our home page

Now we know how to create link URLs, what about adding a small menu in our home for some of the other pages in the site?

```
<p>Please select an option</p>
<ol>
  <li><a href="product/list.html" th:href="@{/product/list}">Product List</a></li>
  <li><a href="order/list.html" th:href="@{/order/list}">Order List</a></li>
  <li><a href="subscribe.html" th:href="@{/subscribe}">Subscribe to our Newsletter</a></li>
  <li><a href="userprofile.html" th:href="@{/userprofile}">See User Profile</a></li>
</ol>
```

4.5 Literals

Text literals

Text literals are just character strings specified between single quotes. They can include any character, but you should escape any single quotes inside them as `\'`.

```
<p>
  Now you are looking at a <span th:text="'working web application'">template file</span>.
</p>
```

Number literals

Numeric literals look exactly like what they are: numbers.

```
<p>The year is <span th:text="2013">1492</span>.</p>
<p>In two years, it will be <span th:text="2013 + 2">1494</span>.</p>
```

Boolean literals

The boolean literals are `true` and `false`. For example:

```
<div th:if="${user.isAdmin()} == false"> ...
```

Note that in the above example, the `== false` is written outside the braces, and thus it is Thymeleaf itself who takes care of it. If it were written inside the braces, it would be the responsibility of the OGNL/SpringEL engines:

```
<div th:if="${user.isAdmin() == false}"> ...
```

The null literal

The `null` literal can be also used:

```
<div th:if="${variable.something} == null"> ...
```

Literal tokens

Numeric, boolean and null literals are in fact a particular case of *literal tokens*.

These tokens allow a little bit of simplification in Standard Expressions. They work exactly the same as text literals (`'...'`), but they only allow letters (`A-Z` and `a-z`), numbers (`0-9`), brackets (`[` and `]`), dots (`.`), hyphens (`-`) and underscores (`_`). So no whitespaces, no commas, etc.

The nice part? Tokens don't need any quotes surrounding them. So we can do this:

```
<div th:class="content">...</div>
```

instead of:

```
<div th:class="'content'">...</div>
```

4.6 Appending texts

Texts, no matter whether they are literals or the result of evaluating variable or message expressions, can be easily appended using the `+` operator:

```
th:text="'The name of the user is ' + ${user.name}"
```

4.7 Literal substitutions

Literal substitutions allow the easy formatting of strings containing values from variables without the need to append literals with `'...' + '...'`.

These substitutions must be surrounded by vertical bars (`|`), like:

```
<span th:text="|Welcome to our application, ${user.name}!|">
```

Which is actually equivalent to:

```
<span th:text="'Welcome to our application, ' + ${user.name} + '!'">
```

Literal substitutions can be combined with other types of expressions:

```
<span th:text="${onevar} + ' ' + |${twovar}, ${threevar}|">
```

Note: only variable expressions (`${...}`) are allowed inside `|...|` literal substitutions. No other literals (`'...'`), boolean/numeric tokens, conditional expressions etc. are.

4.8 Arithmetic operations

Some arithmetic operations are also available: `+`, `-`, `*`, `/` and `%`.

```
th:with="isEven=(${prodStat.count} % 2 == 0)"
```

Note that these operators can also be applied inside OGNL variable expressions themselves (and in that case will be executed by OGNL instead of the Thymeleaf Standard Expression engine):

```
th:with="isEven=${prodStat.count % 2 == 0}"
```

Note that textual aliases exist for some of these operators: `div` (`/`), `mod` (`%`).

4.9 Comparators and Equality

Values in expressions can be compared with the `>`, `<`, `>=` and `<=` symbols, as usual, and also the `==` and `!=` operators can be used to check equality (or the lack of it). Note that XML establishes that the `<` and `>` symbols should not be used in attribute values, and so they should be substituted by `<` and `>`.

```
th:if="${prodStat.count} &gt; 1"
th:text="'Execution mode is ' + ( (${execMode} == 'dev')? 'Development' : 'Production')"
```

Note that textual aliases exist for some of these operators: `gt` (`>`), `lt` (`<`), `ge` (`>=`), `le` (`<=`), `not` (`!`). Also `eq` (`==`), `neq`/`ne` (`!=`).

4.10 Conditional expressions

Conditional expressions are meant to evaluate only one of two expressions depending on the result of evaluating a condition (which is itself another expression).

Let's have a look at an example fragment (introducing another *attribute modifier*, this time `th:class`):

```
<tr th:class="${row.even}? 'even' : 'odd'">
  ...
</tr>
```

All three parts of a conditional expression (`condition`, `then` and `else`) are themselves expressions, which means that they can be variables (`${...}`, `*{...}`), messages (`#{...}`), URLs (`@{...}`) or literals (`'...'`).

Conditional expressions can also be nested using parentheses:

```
<tr th:class="${row.even}? (${row.first}? 'first' : 'even') : 'odd'">
  ...
</tr>
```

Else expressions can also be omitted, in which case a null value is returned if the condition is false:

```
<tr th:class="${row.even}? 'alt'">
```

```
  ...
</tr>
```

4.11 Default expressions (Elvis operator)

A *default expression* is a special kind of conditional value without a *then* part. It is equivalent to the *Elvis operator* present in some languages like Groovy, and allows to specify two expressions, being the second one evaluated only in the case of the first one returning null.

Let's see it in action in our user profile page:

```
<div th:object="${session.user}">
  ...
  <p>Age: <span th:text="*{age}?: '(no age specified)'">27</span>.</p>
</div>
```

As you can see, the operator is `?:`, and we use it here to specify a default value for a name (a literal value, in this case) only if the result of evaluating `*{age}` is null. This is therefore equivalent to:

```
<p>Age: <span th:text="*{age != null}? *{age} : '(no age specified)'">27</span>.</p>
```

As with conditional values, they can contain nested expressions between parentheses:

```
<p>
  Name:
  <span th:text="*{firstName}?: (*{admin}? 'Admin' : #{default.username})">Sebastian</span>
</p>
```

4.12 Preprocessing

In addition to all these features for expression processing, Thymeleaf offers to us the possibility of *preprocessing* expressions.

And what is that preprocessing thing? It is an execution of the expressions done before the normal one, that allows the modification of the actual expression that will be eventually executed.

Preprocessed expressions are exactly like normal ones, but appear surrounded by a double underscore symbol (like `__${expression}__`).

Let's imagine we have an i18n `Messages_fr.properties` entry containing an OGNL expression calling a language-specific static method, like:

```
article.text=@myapp.translator.Translator@translateToFrench({0})
```

...and a `Messages_es.properties equivalent`:

```
article.text=@myapp.translator.Translator@translateToSpanish({0})
```

We can create a fragment of markup that evaluates one expression or the other depending on the locale. For this, we will first select the expression (by preprocessing) and then let Thymeleaf execute it:

```
<p th:text="${__#{article.text('textVar')}__}">Some text here...</p>
```

Note that the preprocessing step for a French locale will be creating the following equivalent:

```
<p th:text="${@myapp.translator.Translator@translateToFrench(textVar)}">Some text here...</p>
```

The preprocessing String `__` can be escaped in attributes using `\_\_`.

5 Setting Attribute Values

This chapter will explain the way in which we can set (or modify) values of attributes in our markup tags, possibly the next most basic feature we will need after setting the tag body content.

5.1 Setting the value of any attribute

Say our website publishes a newsletter, and we want our users to be able to subscribe to it, so we create a `/WEB-INF/templates/subscribe.html` template with a form:

```
<form action="subscribe.html">
  <fieldset>
    <input type="text" name="email" />
    <input type="submit" value="Subscribe me!" />
  </fieldset>
</form>
```

It looks quite OK, but the fact is that this file looks more like a static XHTML page than a template for a web application. First, the action attribute in our form statically links to the template file itself, so that there is no place for useful URL rewriting. Second, the value attribute in the submit button makes it display a text in English, but we'd like it to be internationalized.

Enter then the `th:attr` attribute, and its ability to change the value of attributes of the tags it is set in:

```
<form action="subscribe.html" th:attr="action=@{/subscribe}">
  <fieldset>
    <input type="text" name="email" />
    <input type="submit" value="Subscribe me!" th:attr="value=#{subscribe.submit}"/>
  </fieldset>
</form>
```

The concept is quite straightforward: `th:attr` simply takes an expression that assigns a value to an attribute. Having created the corresponding controller and messages files, the result of processing this file will be as expected:

```
<form action="/gtvg/subscribe">
  <fieldset>
    <input type="text" name="email" />
    <input type="submit" value="¡Suscríbeme!"/>
  </fieldset>
</form>
```

Besides the new attribute values, you can also see that the applicacion context name has been automatically prefixed to the URL base in `/gtvg/subscribe`, as explained in the previous chapter.

But what if we wanted to set more than one attribute at a time? XML rules do not allow you to set an attribute twice in a tag, so `th:attr` will take a comma-separated list of assignments, like:

```
<img src="../../images/gtvglogo.png"
     th:attr="src=@{/images/gtvglogo.png},title=#{logo},alt=#{logo}" />
```

Given the required messages files, this will output:

```
<img src="/gtgv/images/gtvglogo.png" title="Logo de Good Thymes" alt="Logo de Good Thymes" />
```

5.2 Setting value to specific attributes

By now, you might be thinking that something like:

```
<input type="submit" value="Subscribe me!" th:attr="value=#{subscribe.submit}"/>
```

...is quite an ugly piece of markup. Specifying an assignment inside an attribute's value can be very practical, but it is not the most elegant way of creating templates if you have to do it all the time.

Thymeleaf agrees with you. And that's why in fact `th:attr` is scarcely used in templates. Normally, you will be using other `th:*` attributes whose task is setting specific tag attributes (and not just any attribute like `th:attr`).

And which attribute does the Standard Dialect offer us for setting the `value` attribute of our button? Well, in a rather obvious manner, it's `th:value`. Let's have a look:

```
<input type="submit" value="Subscribe me!" th:value="#{subscribe.submit}"/>
```

This looks much better!. Let's try and do the same to the `action` attribute in the `form` tag:

```
<form action="subscribe.html" th:action="@{/subscribe}">
```

And do you remember those `th:href` we put in our `home.html` before? They are exactly this same kind of attributes:

```
<li><a href="product/list.html" th:href="@{/product/list}">Product List</a></li>
```

There are quite a lot of attributes like these, each of them targeting a specific XHTML or HTML5 attribute:

th:abbr	th:accept	th:accept-charset
th:accesskey	th:action	th:align
th:alt	th:archive	th:audio
th:autocomplete	th:axis	th:background
th:bgcolor	th:border	th:cellpadding
th:cellspacing	th:challenge	th:charset
th:cite	th:class	th:classid
th:codebase	th:codetype	th:cols
th:colspan	th:compact	th:content
th:contenteditable	th:contextmenu	th:data
th:datetime	th:dir	th:draggable
th:dropzone	th:enctype	th:for
th:form	th:formaction	th:formenctype
th:formmethod	th:formtarget	th:frame
th:frameborder	th:headers	th:height
th:high	th:href	th:hreflang
th:hspace	th:http-equiv	th:icon
th:id	th:keytype	th:kind
th:label	th:lang	th:list
th:longdesc	th:low	th:manifest
th:marginheight	th:marginwidth	th:max
th:maxlength	th:media	th:method
th:min	th:name	th:optimum
th:pattern	th:placeholder	th:poster
th:preload	th:radiogroup	th:rel
th:rev	th:rows	th:rowspan
th:rules	th:sandbox	th:scheme

th:scope	th:scrolling	th:size
th:sizes	th:span	th:spellcheck
th:src	th:srclang	th:standby
th:start	th:step	th:style
th:summary	th:tabindex	th:target
th:title	th:type	th:usemap
th:value	th:valuetype	th:vspace
th:width	th:wrap	th:xmlbase
th:xmllang	th:xmlspace	

5.3 Setting more than one value at a time

There are two rather special attributes called `th:alt-title` and `th:lang-xmllang` which can be used for setting two attributes to the same value at the same time. Specifically:

- `th:alt-title` will set `alt` and `title`.
- `th:lang-xmllang` will set `lang` and `xml:lang`.

For our GTVG home page, this will allow us to substitute this:

```
<img src="../../images/gtvglogo.png"
     th:attr="src=@{/images/gtvglogo.png},title=#{logo},alt=#{logo}" />
```

...or this, which is equivalent:

```
<img src="../../images/gtvglogo.png"
     th:src="@{/images/gtvglogo.png}" th:title="#{logo}" th:alt="#{logo}" />
```

...by this:

```
<img src="../../images/gtvglogo.png"
     th:src="@{/images/gtvglogo.png}" th:alt-title="#{logo}" />
```

5.4 Appending and prepending

Working in an equivalent way to `th:attr`, Thymeleaf offers the `th:attrappend` and `th:attrprepend` attributes, which append (suffix) or prepend (prefix) the result of their evaluation to the existing attribute values.

For example, you might want to store the name of a CSS class to be added (not set, just added) to one of your buttons in a context variable, because the specific CSS class to be used would depend on something that the user did before. Easy:

```
<input type="button" value="Do it!" class="btn" th:attrappend="class=${' ' + cssStyle}" />
```

If you process this template with the `cssStyle` variable set to `"warning"`, you will get:

```
<input type="button" value="Do it!" class="btn warning" />
```

There are also two specific *appending attributes* in the Standard Dialect: the `th:classappend` and `th:styleappend` attributes, which are used for adding a CSS class or a fragment of *style* to an element without overwriting the existing ones:

```
<tr th:each="prod : ${prods}" class="row" th:classappend="${prodStat.odd}? 'odd'">
```

(Don't worry about that `th:each` attribute. It is an *iterating attribute* and we will talk about it later.)

5.5 Fixed-value boolean attributes

Some XHTML/HTML5 attributes are special in that, either they are present in their elements with a specific and fixed value, or they are not present at all.

For example, `checked`:

```
<input type="checkbox" name="option1" checked="checked" />
<input type="checkbox" name="option2" />
```

No other value than `"checked"` is allowed according to the XHTML standards for the `checked` attribute (HTML5 rules are a little more relaxed on that). And the same happens with `disabled`, `multiple`, `readonly` and `selected`.

The Standard Dialect includes attributes that allow you to set these attributes by evaluating a condition, so that if evaluated to true, the attribute will be set to its fixed value, and if evaluated to false, the attribute will not be set:

```
<input type="checkbox" name="active" th:checked="${user.active}" />
```

The following fixed-value boolean attributes exist in the Standard Dialect:

th:async	th:autofocus	th:autoplay
th:checked	th:controls	th:declare
th:default	th:defer	th:disabled
th:formnovalidate	th:hidden	th:ismap
th:loop	th:multiple	th:novalidate
th:nowrap	th:open	th:pubdate
th:readonly	th:required	th:reversed
th:scoped	th:seamless	th:selected

5.6 Support for HTML5-friendly attribute and element names

It is also possible to use a completely different syntax to apply processors to your templates, more HTML5-friendly.

```
<table>
    <tr data-th-each="user : ${users}">
        <td data-th-text="${user.login}">...</td>
        <td data-th-text="${user.name}">...</td>
    </tr>
</table>
```

The `data-{prefix}-{name}` syntax is the standard way to write custom attributes in HTML5, without requiring developers to use any namespaced names like `th:*`. Thymeleaf makes this syntax automatically available to all your dialects (not only the Standard ones).

There is also a syntax to specify custom tags: `{prefix}-{name}`, which follows the *W3C Custom Elements specification* (a part of the larger *W3C Web Components spec*). This can be used, for example, for the `th:block` element (or also `th-block`), which will be explained in a later section.

Important: this syntax is an addition to the namespaced `th:*` one, it does not replace it. There is no intention at all to deprecate the namespaced syntax in the future.

6 Iteration

So far we have created a home page, a user profile page and also a page for letting users subscribe to our newsletter... but what about our products? Shouldn't we build a product list to let visitors know what we sell? Well, obviously yes. And there we go now.

6.1 Iteration basics

For listing our products in our `/WEB-INF/templates/product/list.html` page we will need a table. Each of our products will be displayed in a row (a `<tr>` element), and so for our template we will need to create a *template row* — one that will exemplify how we want each product to be displayed— and then instruct Thymeleaf to *iterate it* once for each product.

The Standard Dialect offers us an attribute for exactly that, `th:each`.

Using th:each

For our product list page, we will need a controller that retrieves the list of products from the service layer and adds it to the template context:

```
public void process(
        HttpServletRequest request, HttpServletResponse response,
        ServletContext servletContext, TemplateEngine templateEngine) {

    ProductService productService = new ProductService();
    List<Product> allProducts = productService.findAll();

    WebContext ctx = new WebContext(request, servletContext, request.getLocale());
    ctx.setVariable("prods", allProducts);

    templateEngine.process("product/list", ctx, response.getWriter());
}
```

And then we will use `th:each` in our template to iterate the list of products:

```
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-4.dtd">

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

  <head>
    <title>Good Thymes Virtual Grocery</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <link rel="stylesheet" type="text/css" media="all"
          href="../../../css/gtvg.css" th:href="@{/css/gtvg.css}" />
  </head>

  <body>

    <h1>Product list</h1>

    <table>
      <tr>
        <th>NAME</th>
        <th>PRICE</th>
        <th>IN STOCK</th>
      </tr>
      <tr th:each="prod : ${prods}">
        <td th:text="${prod.name}">Onions</td>
        <td th:text="${prod.price}">2.41</td>
        <td th:text="${prod.inStock}? #{true} : #{false}">yes</td>
      </tr>
    </table>
```

```
    <p>
      <a href="../home.html" th:href="@{/}">Return to home</a>
    </p>

  </body>

</html>
```

That `prod : ${prods}` attribute value you see above means "for each element in the result of evaluating `${prods}`, repeat this fragment of template setting that element into a variable called prod". Let's give a name each of the things we see:

- We will call `${prods}` the *iterated expression* or *iterated variable*.
- We will call `prod` the *iteration variable* or simply *iter variable*.

Note that the `prod` iter variable will only be available inside the `<tr>` element (including inner tags like `<td>`).

Iterable values

Not only `java.util.List` objects can be used for iteration in Thymeleaf. In fact, there is a quite complete set of objects that are considered *iterable* by a `th:each` attribute:

- Any object implementing `java.util.Iterable`
- Any object implementing `java.util.Map`. When iterating maps, iter variables will be of class `java.util.Map.Entry`.
- Any array
- Any other object will be treated as if it were a single-valued list containing the object itself.

6.2 Keeping iteration status

When using `th:each`, Thymeleaf offers a mechanism useful for keeping track of the status of your iteration: the *status variable*.

Status variables are defined within a `th:each` attribute and contain the following data:

- The current *iteration index*, starting with 0. This is the `index` property.
- The current *iteration index*, starting with 1. This is the `count` property.
- The total amount of elements in the iterated variable. This is the `size` property.
- The *iter variable* for each iteration. This is the `current` property.
- Whether the current iteration is even or odd. These are the `even/odd` boolean properties.
- Whether the current iteration is the first one. This is the `first` boolean property.
- Whether the current iteration is the last one. This is the `last` boolean property.

Let's see how we could use it within the previous example:

```
<table>
  <tr>
    <th>NAME</th>
    <th>PRICE</th>
    <th>IN STOCK</th>
  </tr>
  <tr th:each="prod,iterStat : ${prods}" th:class="${iterStat.odd}? 'odd'">
    <td th:text="${prod.name}">Onions</td>
    <td th:text="${prod.price}">2.41</td>
    <td th:text="${prod.inStock}? #{true} : #{false}">yes</td>
  </tr>
</table>
```

As you can see, the status variable (`iterStat` in this example) is defined in the `th:each` attribute by writing its name after the iter variable itself, separated by a comma. As happens to the iter variable, the status variable will only be

available inside the fragment of code defined by the tag holding the `th:each` attribute.

Let's have a look at the result of processing 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">

  <head>
    <title>Good Thymes Virtual Grocery</title>
    <meta content="text/html; charset=UTF-8" http-equiv="Content-Type"/>
    <link rel="stylesheet" type="text/css" media="all" href="/gtvg/css/gtvg.css" />
  </head>

  <body>

    <h1>Product list</h1>

    <table>
      <tr>
        <th colspan="1" rowspan="1">NAME</th>
        <th colspan="1" rowspan="1">PRICE</th>
        <th colspan="1" rowspan="1">IN STOCK</th>
      </tr>
      <tr>
        <td colspan="1" rowspan="1">Fresh Sweet Basil</td>
        <td colspan="1" rowspan="1">4.99</td>
        <td colspan="1" rowspan="1">yes</td>
      </tr>
      <tr class="odd">
        <td colspan="1" rowspan="1">Italian Tomato</td>
        <td colspan="1" rowspan="1">1.25</td>
        <td colspan="1" rowspan="1">no</td>
      </tr>
      <tr>
        <td colspan="1" rowspan="1">Yellow Bell Pepper</td>
        <td colspan="1" rowspan="1">2.50</td>
        <td colspan="1" rowspan="1">yes</td>
      </tr>
      <tr class="odd">
        <td colspan="1" rowspan="1">Old Cheddar</td>
        <td colspan="1" rowspan="1">18.75</td>
        <td colspan="1" rowspan="1">yes</td>
      </tr>
    </table>

    <p>
      <a href="/gtvg/" shape="rect">Return to home</a>
    </p>

  </body>

</html>
```

Note that our iteration status variable has worked perfectly, establishing the `odd` CSS class only to odd rows (row counting starts with 0).

> All those colspan and rowspan attributes in the `<td>` tags, as well as the shape one in `<a>` are automatically added by Thymeleaf in accordance with the DTD for the selected *XHTML 1.0 Strict* standard, that establishes those values as default for those attributes (remember that our template didn't set a value for them). Don't worry about them at all, because they will not affect the display of your page. As an example, if we were using HTML5 (which has no DTD), those attributes would never be added.

If you don't explicitly set a status variable, Thymeleaf will always create one for you by suffixing `Stat` to the name of the iteration variable:

```
<table>
  <tr>
```

```
    <th>NAME</th>
    <th>PRICE</th>
    <th>IN STOCK</th>
  </tr>
  <tr th:each="prod : ${prods}" th:class="${prodStat.odd}? 'odd'">
    <td th:text="${prod.name}">Onions</td>
    <td th:text="${prod.price}">2.41</td>
    <td th:text="${prod.inStock}? #{true} : #{false}">yes</td>
  </tr>
</table>
```

7 Conditional Evaluation

7.1 Simple conditionals: “if” and “unless”

Sometimes you will need a fragment of your template only to appear in the result if a certain condition is met.

For example, imagine we want to show in our product table a column with the number of comments that exist for each product and, if there are any comments, a link to the comment detail page for that product.

In order to do this, we would use the `th:if` attribute:

```
<table>
  <tr>
    <th>NAME</th>
    <th>PRICE</th>
    <th>IN STOCK</th>
    <th>COMMENTS</th>
  </tr>
  <tr th:each="prod : ${prods}" th:class="${prodStat.odd}? 'odd'">
    <td th:text="${prod.name}">Onions</td>
    <td th:text="${prod.price}">2.41</td>
    <td th:text="${prod.inStock}? #{true} : #{false}">yes</td>
    <td>
      <span th:text="${#lists.size(prod.comments)}">2</span> comment/s
      <a href="comments.html"
         th:href="@{/product/comments(prodId=${prod.id})}"
         th:if="${not #lists.isEmpty(prod.comments)}">view</a>
    </td>
  </tr>
</table>
```

Quite a lot of things to see here, so let’s focus on the important line:

```
<a href="comments.html"
   th:href="@{/product/comments(prodId=${prod.id})}"
   th:if="${not #lists.isEmpty(prod.comments)}">view</a>
```

There is little to explain from this code, in fact: We will be creating a link to the comments page (with URL `/product/comments`) with a `prodId` parameter set to the `id` of the product, but only if the product has any comments.

Let’s have a look at the resulting markup (getting rid of the defaulted `rowspan` and `colspan` attributes for a cleaner view):

```
<table>
  <tr>
    <th>NAME</th>
    <th>PRICE</th>
    <th>IN STOCK</th>
    <th>COMMENTS</th>
  </tr>
  <tr>
    <td>Fresh Sweet Basil</td>
    <td>4.99</td>
    <td>yes</td>
    <td>
      <span>0</span> comment/s
    </td>
  </tr>
  <tr class="odd">
    <td>Italian Tomato</td>
    <td>1.25</td>
    <td>no</td>
    <td>
```

```
      <span>2</span> comment/s
      <a href="/gtvg/product/comments?prodId=2">view</a>
    </td>
  </tr>
  <tr>
    <td>Yellow Bell Pepper</td>
    <td>2.50</td>
    <td>yes</td>
    <td>
      <span>0</span> comment/s
    </td>
  </tr>
  <tr class="odd">
    <td>Old Cheddar</td>
    <td>18.75</td>
    <td>yes</td>
    <td>
      <span>1</span> comment/s
      <a href="/gtvg/product/comments?prodId=4">view</a>
    </td>
  </tr>
</table>
```

Perfect! That's exactly what we wanted.

Note that the `th:if` attribute will not only evaluate *boolean* conditions. Its capabilities go a little beyond that, and it will evaluate the specified expression as `true` following these rules:

- If value is not null:
  - If value is a boolean and is `true`.
  - If value is a number and is non-zero
  - If value is a character and is non-zero
  - If value is a String and is not "false", "off" or "no"
  - If value is not a boolean, a number, a character or a String.
- (If value is null, th:if will evaluate to false).

Also, `th:if` has a negative counterpart, `th:unless`, which we could have used in the previous example instead of using a `not` inside the OGNL expression:

```
<a href="comments.html"
   th:href="@{/comments(prodId=${prod.id})}"
   th:unless="${#lists.isEmpty(prod.comments)}">view</a>
```

7.2 Switch statements

There is also a way to display content conditionally using the equivalent of a *switch* structure in Java: the `th:switch` / `th:case` attribute set.

They work exactly as you would expect:

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

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

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

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

```
  <p th:case="*">User is some other thing</p>
</div>
```

8 Template Layout

8.1 Including template fragments

Defining and referencing fragments

We will often want to include in our templates fragments from other templates. Common uses for this are footers, headers, menus…

In order to do this, Thymeleaf needs us to define the fragments available for inclusion, which we can do by using the `th:fragment` attribute.

Now let's say we want to add a standard copyright footer to all our grocery pages, and for that we define a `/WEB-INF/templates/footer.html` file containing this code:

```
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-4.dtd">

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

  <body>

    <div th:fragment="copy">
      &copy; 2011 The Good Thymes Virtual Grocery
    </div>

  </body>

</html>
```

The code above defines a fragment called `copy` that we can easily include in our home page using one of the `th:include` or `th:replace` attributes:

```
<body>

  ...

  <div th:include="footer :: copy"></div>

</body>
```

The syntax for both these inclusion attributes is quite straightforward. There are three different formats:

- `"templatename::domselector"` or the equivalent `templatename::[domselector]` Includes the fragment resulting from executing the specified DOM Selector on the template named `templatename`.
    - Note that `domselector` can be a mere fragment name, so you could specify something as simple as `templatename::fragmentname` like in the `footer :: copy` above.

    > DOM Selector syntax is similar to XPath expressions and CSS selectors, see the Appendix C for more info on this syntax.

- `"templatename"` Includes the complete template named `templatename`.

    > Note that the template name you use in `th:include`/`th:replace` tags will have to be resolvable by the Template Resolver currently being used by the Template Engine.

- `::domselector"` or `"this::domselector"` Includes a fragment from the same template.

Both `templatename` and `domselector` in the above examples can be fully-featured expressions (even conditionals!) like:

```
<div th:include="footer :: (${user.isAdmin}? #{footer.admin} : #{footer.normaluser})"></div>
```

Fragments can include any `th:* attributes`. These attributes will be evaluated once the fragment is included into the target template (the one with the `th:include` / `th:replace` attribute), and they will be able to reference any context variables defined in this target template.

> A big advantage of this approach to fragments is that you can write your fragments' code in pages that are perfectly displayable by a browser, with a complete and even validating XHTML structure, while still retaining the ability to make Thymeleaf include them into other templates.

Referencing fragments without `th:fragment`

Besides, thanks to the power of DOM Selectors, we can include fragments that do not use any `th:fragment` attributes. It can even be markup code coming from a different application with no knowledge of Thymeleaf at all:

```
...
<div id="copy-section">
  &copy; 2011 The Good Thymes Virtual Grocery
</div>
...
```

We can use the fragment above simply referencing it by its `id` attribute, in a similar way to a CSS selector:

```
<body>

  ...

  <div th:include="footer :: #copy-section"></div>

</body>
```

Difference between `th:include` and `th:replace`

And what is the difference between `th:include` and `th:replace`? Whereas `th:include` will include the contents of the fragment into its host tag, `th:replace` will actually substitute the host tag by the fragment's. So that an HTML5 fragment like this:

```
<footer th:fragment="copy">
  &copy; 2011 The Good Thymes Virtual Grocery
</footer>
```

...included twice in host `<div>` tags, like this:

```
<body>

  ...

  <div th:include="footer :: copy"></div>
  <div th:replace="footer :: copy"></div>

</body>
```

...will result in:

```
<body>
```

```
  ...

  <div>
    &copy; 2011 The Good Thymes Virtual Grocery
  </div>
  <footer>
    &copy; 2011 The Good Thymes Virtual Grocery
  </footer>

</body>
```

The `th:substituteby` attribute can also be used as an alias for `th:replace`, but the latter is recommended. Note that `th:substituteby` might be deprecated in future versions.

8.2 Parameterizable fragment signatures

In order to create a more *function-like* mechanism for the use of template fragments, fragments defined with `th:fragment` can specify a set of parameters:

```
<div th:fragment="frag (onevar,twovar)">
    <p th:text="${onevar} + ' - ' + ${twovar}">...</p>
</div>
```

This requires the use of one of these two syntaxes to call the fragment from `th:include`, `th:replace`:

```
<div th:include="::frag (${value1},${value2})">...</div>
<div th:include="::frag (onevar=${value1},twovar=${value2})">...</div>
```

Note that order is not important in the last option:

```
<div th:include="::frag (twovar=${value2},onevar=${value1})">...</div>
```

Fragment local variables without fragment signature

Even if fragments are defined without signature, like this:

```
<div th:fragment="frag">
    ...
</div>
```

We could use the second syntax specified above to call them (and only the second one):

```
<div th:include="::frag (onevar=${value1},twovar=${value2})">
```

This would be, in fact, equivalent to a combination of `th:include` and `th:with`:

```
<div th:include="::frag" th:with="onevar=${value1},twovar=${value2}">
```

Note that this specification of local variables for a fragment —no matter whether it has a signature or not— does not cause the context to emptied previously to its execution. Fragments will still be able to access every context variable being used at the calling template like they currently are.

th:assert for in-template assertions

The `th:assert` attribute can specify a comma-separated list of expressions which should be evaluated and produce true for every evaluation, raising an exception if not.

```
<div th:assert="${onevar},(${twovar} != 43)">...</div>
```

This comes in handy for validating parameters at a fragment signature:

```
<header th:fragment="contentheader(title)"
th:assert="${!#strings.isEmpty(title)}">...</header>
```

8.3 Removing template fragments

Let's revisit the last version of our product list template:

```
<table>
  <tr>
    <th>NAME</th>
    <th>PRICE</th>
    <th>IN STOCK</th>
    <th>COMMENTS</th>
  </tr>
  <tr th:each="prod : ${prods}" th:class="${prodStat.odd}? 'odd'">
    <td th:text="${prod.name}">Onions</td>
    <td th:text="${prod.price}">2.41</td>
    <td th:text="${prod.inStock}? #{true} : #{false}">yes</td>
    <td>
      <span th:text="${#lists.size(prod.comments)}">2</span> comment/s
      <a href="comments.html"
         th:href="@{/product/comments(prodId=${prod.id})}"
         th:unless="${#lists.isEmpty(prod.comments)}">view</a>
    </td>
  </tr>
</table>
```

This code is just fine as a template, but as a static page (when directly open by a browser without Thymeleaf processing it) it would not make a nice prototype.

Why? Because although perfectly displayable by browsers, that table only has a row, and this row has mock data. As a prototype, it simply wouldn't look realistic enough... we should have more than one product, *we need more rows*.

So let's add some:

```
<table>
  <tr>
    <th>NAME</th>
    <th>PRICE</th>
    <th>IN STOCK</th>
    <th>COMMENTS</th>
  </tr>
  <tr th:each="prod : ${prods}" th:class="${prodStat.odd}? 'odd'">
    <td th:text="${prod.name}">Onions</td>
    <td th:text="${prod.price}">2.41</td>
    <td th:text="${prod.inStock}? #{true} : #{false}">yes</td>
    <td>
      <span th:text="${#lists.size(prod.comments)}">2</span> comment/s
      <a href="comments.html"
         th:href="@{/product/comments(prodId=${prod.id})}"
         th:unless="${#lists.isEmpty(prod.comments)}">view</a>
    </td>
  </tr>
  <tr class="odd">
    <td>Blue Lettuce</td>
    <td>9.55</td>
    <td>no</td>
    <td>
      <span>0</span> comment/s
    </td>
  </tr>
  <tr>
    <td>Mild Cinnamon</td>
```

```
    <td>1.99</td>
    <td>yes</td>
    <td>
      <span>3</span> comment/s
      <a href="comments.html">view</a>
    </td>
  </tr>
</table>
```

Ok, now we have three, definitely better for a prototype. But... what will happen when we process it with Thymeleaf?:

```
<table>
  <tr>
    <th>NAME</th>
    <th>PRICE</th>
    <th>IN STOCK</th>
    <th>COMMENTS</th>
  </tr>
  <tr>
    <td>Fresh Sweet Basil</td>
    <td>4.99</td>
    <td>yes</td>
    <td>
      <span>0</span> comment/s
    </td>
  </tr>
  <tr class="odd">
    <td>Italian Tomato</td>
    <td>1.25</td>
    <td>no</td>
    <td>
      <span>2</span> comment/s
      <a href="/gtvg/product/comments?prodId=2">view</a>
    </td>
  </tr>
  <tr>
    <td>Yellow Bell Pepper</td>
    <td>2.50</td>
    <td>yes</td>
    <td>
      <span>0</span> comment/s
    </td>
  </tr>
  <tr class="odd">
    <td>Old Cheddar</td>
    <td>18.75</td>
    <td>yes</td>
    <td>
      <span>1</span> comment/s
      <a href="/gtvg/product/comments?prodId=4">view</a>
    </td>
  </tr>
  <tr class="odd">
    <td>Blue Lettuce</td>
    <td>9.55</td>
    <td>no</td>
    <td>
      <span>0</span> comment/s
    </td>
  </tr>
  <tr>
    <td>Mild Cinnamon</td>
    <td>1.99</td>
    <td>yes</td>
    <td>
      <span>3</span> comment/s
      <a href="comments.html">view</a>
    </td>
  </tr>
</table>
```

The last two rows are mock rows! Well, of course they are: iteration was only applied to the first row, so there is no reason why Thymeleaf should have removed the other two.

We need a way to remove those two rows during template processing. Let's use the `th:remove` attribute on the second and third `<tr>` tags:

```
<table>
  <tr>
    <th>NAME</th>
    <th>PRICE</th>
    <th>IN STOCK</th>
    <th>COMMENTS</th>
  </tr>
  <tr th:each="prod : ${prods}" th:class="${prodStat.odd}? 'odd'">
    <td th:text="${prod.name}">Onions</td>
    <td th:text="${prod.price}">2.41</td>
    <td th:text="${prod.inStock}? #{true} : #{false}">yes</td>
    <td>
      <span th:text="${#lists.size(prod.comments)}">2</span> comment/s
      <a href="comments.html"
         th:href="@{/product/comments(prodId=${prod.id})}"
         th:unless="${#lists.isEmpty(prod.comments)}">view</a>
    </td>
  </tr>
  <tr class="odd" th:remove="all">
    <td>Blue Lettuce</td>
    <td>9.55</td>
    <td>no</td>
    <td>
      <span>0</span> comment/s
    </td>
  </tr>
  <tr th:remove="all">
    <td>Mild Cinnamon</td>
    <td>1.99</td>
    <td>yes</td>
    <td>
      <span>3</span> comment/s
      <a href="comments.html">view</a>
    </td>
  </tr>
</table>
```

Once processed, everything will look again as it should:

```
<table>
  <tr>
    <th>NAME</th>
    <th>PRICE</th>
    <th>IN STOCK</th>
    <th>COMMENTS</th>
  </tr>
  <tr>
    <td>Fresh Sweet Basil</td>
    <td>4.99</td>
    <td>yes</td>
    <td>
      <span>0</span> comment/s
    </td>
  </tr>
  <tr class="odd">
    <td>Italian Tomato</td>
    <td>1.25</td>
    <td>no</td>
    <td>
      <span>2</span> comment/s
      <a href="/gtvg/product/comments?prodId=2">view</a>
    </td>
  </tr>
  <tr>
    <td>Yellow Bell Pepper</td>
    <td>2.50</td>
    <td>yes</td>
    <td>
      <span>0</span> comment/s
    </td>
```

```
    </tr>
    <tr class="odd">
      <td>Old Cheddar</td>
      <td>18.75</td>
      <td>yes</td>
      <td>
        <span>1</span> comment/s
        <a href="/gtvg/product/comments?prodId=4">view</a>
      </td>
    </tr>
  </table>
```

And what about that `all` value in the attribute, what does it mean? Well, in fact `th:remove` can behave in five different ways, depending on its value:

- `all`: Remove both the containing tag and all its children.
- `body`: Do not remove the containing tag, but remove all its children.
- `tag`: Remove the containing tag, but do not remove its children.
- `all-but-first`: Remove all children of the containing tag except the first one.
- `none` : Do nothing. This value is useful for dynamic evaluation.

What can that `all-but-first` value be useful for? It will let us save some `th:remove="all"` when prototyping:

```
<table>
  <thead>
    <tr>
      <th>NAME</th>
      <th>PRICE</th>
      <th>IN STOCK</th>
      <th>COMMENTS</th>
    </tr>
  </thead>
  <tbody th:remove="all-but-first">
    <tr th:each="prod : ${prods}" th:class="${prodStat.odd}? 'odd'">
      <td th:text="${prod.name}">Onions</td>
      <td th:text="${prod.price}">2.41</td>
      <td th:text="${prod.inStock}? #{true} : #{false}">yes</td>
      <td>
        <span th:text="${#lists.size(prod.comments)}">2</span> comment/s
        <a href="comments.html"
           th:href="@{/product/comments(prodId=${prod.id})}"
           th:unless="${#lists.isEmpty(prod.comments)}">view</a>
      </td>
    </tr>
    <tr class="odd">
      <td>Blue Lettuce</td>
      <td>9.55</td>
      <td>no</td>
      <td>
        <span>0</span> comment/s
      </td>
    </tr>
    <tr>
      <td>Mild Cinnamon</td>
      <td>1.99</td>
      <td>yes</td>
      <td>
        <span>3</span> comment/s
        <a href="comments.html">view</a>
      </td>
    </tr>
  </tbody>
</table>
```

The `th:remove` attribute can take any *Thymeleaf Standard Expression*, as long as it returns one of the allowed String values (`all`, `tag`, `body`, `all-but-first` or `none`).

This means removals could be conditional, like:

```
<a href="/something" th:remove="${condition}? tag : none">Link text not to be removed</a>
```

Also note that `th:remove` considers `null` a synonym to `none`, so that the following works exactly as the example above:

```
<a href="/something" th:remove="${condition}? tag">Link text not to be removed</a>
```

In this case, if `${condition}` is false, `null` will be returned, and thus no removal will be performed.

9 Local Variables

Thymeleaf calls *local variables* those variables that are defined for a specific fragment of a template, and are only available for evaluation inside that fragment.

An example we have already seen is the `prod` iter variable in our product list page:

```
<tr th:each="prod : ${prods}">
    ...
</tr>
```

That `prod` variable will be available only within the bonds of the `<tr>` tag. Specifically:

- It will be available for any other `th:*` attributes executing in that tag with less *precedence* than `th:each` (which means they will execute after `th:each`).
- It will be available for any child element of the `<tr>` tag, such as `<td>` elements.

Thymeleaf offers you a way to declare local variables without iteration. It is the `th:with` attribute, and its syntax is like that of attribute value assignments:

```
<div th:with="firstPer=${persons[0]}">
  <p>
    The name of the first person is <span th:text="${firstPer.name}">Julius Caesar</span>.
  </p>
</div>
```

When `th:with` is processed, that `firstPer` variable is created as a local variable and added to the variables map coming from the context, so that it is as available for evaluation as any other variables declared in the context from the beginning, but only within the bounds of the containing `<div>` tag.

You can define several variables at the same time using the usual multiple assignment syntax:

```
<div th:with="firstPer=${persons[0]},secondPer=${persons[1]}">
  <p>
    The name of the first person is <span th:text="${firstPer.name}">Julius Caesar</span>.
  </p>
  <p>
    But the name of the second person is
    <span th:text="${secondPer.name}">Marcus Antonius</span>.
  </p>
</div>
```

The `th:with` attribute allows reusing variables defined in the same attribute:

```
<div th:with="company=${user.company + ' Co.'},account=${accounts[company]}">...</div>
```

Let's use this in our Grocery's home page! Remember the code we wrote for outputting a formatted date?

```
<p>
  Today is:
  <span th:text="${#calendars.format(today,'dd MMMM yyyy')}">13 february 2011</span>
</p>
```

Well, what if we wanted that `"dd MMMM yyyy"` to actually depend on the locale? For example, we might want to add the following message to our `home_en.properties`:

```
date.format=MMMM dd'','' yyyy
```

...and an equivalent one to our `home_es.properties`:

```
date.format=dd ''de'' MMMM'','' yyyy
```

Now, let's use `th:with` to get the localized date format into a variable, and then use it in our `th:text` expression:

```
<p th:with="df=#{date.format}">
  Today is: <span th:text="${#calendars.format(today,df)}">13 February 2011</span>
</p>
```

That was clean and easy. In fact, given the fact that `th:with` has a higher `precedence` than `th:text`, we could have solved this all in the `span` tag:

```
<p>
  Today is:
  <span th:with="df=#{date.format}"
        th:text="${#calendars.format(today,df)}">13 February 2011</span>
</p>
```

You might be thinking: Precedence? We haven't talked about that yet! Well, don't worry because that is exactly what the next chapter is about.

10 Attribute Precedence

What happens when you write more than one `th:*` attribute in the same tag? For example:

```
<ul>
  <li th:each="item : ${items}" th:text="${item.description}">Item description here...</li>
</ul>
```

Of course, we would expect that `th:each` attribute to execute before the `th:text` so that we get the results we want, but given the fact that the DOM (Document Object Model) standard does not give any kind of meaning to the order in which the attributes of a tag are written, a *precedence* mechanism has to be established in the attributes themselves in order to be sure that this will work as expected.

So, all Thymeleaf attributes define a numeric precedence, which establishes the order in which they are executed in the tag. This order is:

Order	Feature	Attributes
1	Fragment inclusion	`th:include` `th:replace`
2	Fragment iteration	`th:each`

3	Conditional evaluation	`th:if`
		`th:unless`
		`th:switch`
		`th:case`

4	Local variable definition	`th:object` `th:with`

5	General attribute modification	`th:attr`
		`th:attrprepend`
		`th:attrappend`

6	Specific attribute modification	`th:value`
		`th:href`
		`th:src`
		`...`

7	Text (tag body modification)	`th:text` `th:utext`
8	Fragment specification	`th:fragment`
9	Fragment removal	`th:remove`

This precedence mechanism means that the above iteration fragment will give exactly the same results if the attribute position is inverted (although it would be slightly less readable):

```
<ul>
  <li th:text="${item.description}" th:each="item : ${items}">Item description here...</li>
</ul>
```

11. Comments and Blocks

11.1. Standard HTML/XML comments

Standard HTML/XML comments `<!-- ... -->` can be used anywhere in thymeleaf templates. Anything inside these comments won't be processed by neither Thymeleaf nor the browser, and will be just copied verbatim to the result:

```
<!-- User info follows -->
<div th:text="${...}">
  ...
</div>
```

11.2. Thymeleaf parser-level comment blocks

Parser-level comment blocks are code that will be simply removed from the template when thymeleaf parses it. They look like this:

```
<!--/* This code will be removed at thymeleaf parsing time! */-->
```

Thymeleaf will remove absolutely everything between `<!--/*` and `*/-->`, so these comment blocks can also be used for displaying code when a template is statically open, knowing that it will be removed when thymeleaf processes it:

```
<!--/*-->
  <div>
    you can see me only before thymeleaf processes me!
  </div>
<!--*/-->
```

This might come very handy for prototyping tables with a lot of `<tr>`'s, for example:

```
<table>
   <tr th:each="x : ${xs}">
     ...
   </tr>
   <!--/*-->
   <tr>
     ...
   </tr>
   <tr>
     ...
   </tr>
   <!--*/-->
</table>
```

11.3. Thymeleaf prototype-only comment blocks

Thymeleaf allows the definition of special comment blocks marked to be comments when the template is open statically (i.e. as a prototype), but considered normal markup by Thymeleaf when executing the template.

```
<span>hello!</span>
<!--/*/
  <div th:text="${...}">
    ...
  </div>
/*/-->
<span>goodbye!</span>
```

Thymeleaf's parsing system will simply remove the `<!--/*/` and `/*/-->` markers, but not its contents, which will be left therefore uncommented. So when executing the template, Thymeleaf will actually see this:

```
<span>hello!</span>

  <div th:text="${...}">
    ...
  </div>

<span>goodbye!</span>
```

As happens with parser-level comment blocks, note that this feature is dialect-independent.

11.4. Synthetic `th:block` tag

Thymeleaf's only element processor (not an attribute) included in the Standard Dialects is `th:block`.

`th:block` is a mere attribute container that allows template developers to specify whichever attributes they want. Thymeleaf will execute these attributes and then simply make the block dissapear without a trace.

So it could be useful, for example, when creating iterated tables that require more than one `<tr>` for each element:

```
<table>
  <th:block th:each="user : ${users}">
    <tr>
        <td th:text="${user.login}">...</td>
        <td th:text="${user.name}">...</td>
    </tr>
    <tr>
        <td colspan="2" th:text="${user.address}">...</td>
    </tr>
  </th:block>
</table>
```

And especially useful when used in combination with prototype-only comment blocks:

```
<table>
    <!--/*/ <th:block th:each="user : ${users}"> /*/-->
    <tr>
        <td th:text="${user.login}">...</td>
        <td th:text="${user.name}">...</td>
    </tr>
    <tr>
        <td colspan="2" th:text="${user.address}">...</td>
    </tr>
    <!--/*/ </th:block> /*/-->
</table>
```

Note how this solution allows templates to be valid HTML (no need to add forbidden `<div>` blocks inside `<table>`), and still works OK when open statically in browsers as prototypes!

12 Inlining

12.1 Text inlining

Although the Standard Dialect allows us to do almost everything we might need by using tag attributes, there are situations in which we could prefer writing expressions directly into our HTML texts. For example, we could prefer writing this:

```
<p>Hello, [[${session.user.name}]]!</p>
```

...instead of this:

```
<p>Hello, <span th:text="${session.user.name}">Sebastian</span>!</p>
```

Expressions between `[[...]]` are considered expression inlining in Thymeleaf, and in them you can use any kind of expression that would also be valid in a `th:text` attribute.

In order for inlining to work, we must activate it by using the `th:inline` attribute, which has three possible values or modes (`text`, `javascript` and `none`). Let's try `text`:

```
<p th:inline="text">Hello, [[${session.user.name}]]!</p>
```

The tag holding the `th:inline` does not have to be the one containing the inlined expression/s, any parent tag would do:

```
<body th:inline="text">
   ...
   <p>Hello, [[${session.user.name}]]!</p>
   ...
</body>
```

So you might now be asking: *Why aren't we doing this from the beginning? It's less code than all those* `th:text` *attributes!* Well, be careful there, because although you might find inlining quite interesting, you should always remember that inlined expressions will be displayed verbatim in your HTML files when you open them statically, so you probably won't be able to use them as prototypes anymore!

The difference between how a browser would statically display our fragment of code without using inlining...

```
Hello, Sebastian!
```

...and using it...

```
Hello, [[${session.user.name}]]!
```

...is quite clear.

12.2 Script inlining (JavaScript and Dart)

Thymeleaf offers a series of "scripting" modes for its inlining capabilities, so that you can integrate your data inside scripts created in some script languages.

Current scripting modes are `javascript` (`th:inline="javascript"`) and `dart` (`th:inline="dart"`).

The first thing we can do with script inlining is writing the value of expressions into our scripts, like:

```
<script th:inline="javascript">
/*<![CDATA[*/
    ...

    var username = /*[[${session.user.name}]]*/ 'Sebastian';

    ...
/*]]>*/
</script>
```

The `/*[[...]]*/` syntax, instructs Thymeleaf to evaluate the contained expression. But there are more implications here:

- Being a javascript comment (`/*...*/`), our expression will be ignored when displaying the page statically in a browser.
- The code after the inline expression (`'Sebastian'`) will be executed when displaying the page statically.
- Thymeleaf will execute the expression and insert the result, but it will also remove all the code in the line after the inline expression itself (the part that is executed when displayed statically).

So, the result of executing this will be:

```
<script th:inline="javascript">
/*<![CDATA[*/
    ...

    var username = 'John Apricot';

    ...
/*]]>*/
</script>
```

You can also do it without comments with the same effects, but that will make your script to fail when loaded statically:

```
<script th:inline="javascript">
/*<![CDATA[*/
    ...

    var username = [[${session.user.name}]];

    ...
/*]]>*/
</script>
```

Note that this evaluation is intelligent and not limited to Strings. Thymeleaf will correctly write in Javascript/Dart syntax the following kinds of objects:

- Strings
- Numbers
- Booleans
- Arrays
- Collections
- Maps
- Beans (objects with *getter* and *setter* methods)

For example, if we had the following code:

```
<script th:inline="javascript">
/*<![CDATA[*/
    ...
```

```
    var user = /*[[${session.user}]]*/ null;

    ...
/*]]>*/
</script>
```

That `${session.user}` expression will evaluate to a `User` object, and Thymeleaf will correctly convert it to Javascript syntax:

```
<script th:inline="javascript">
/*<![CDATA[*/
    ...

    var user = {'age':null,'firstName':'John','lastName':'Apricot',
                'name':'John Apricot','nationality':'Antarctica'};

    ...
/*]]>*/
</script>
```

Adding code

An additional feature when using javascript inlining is the ability to include code between a special comment syntax `/*[+...+]*/` so that Thymeleaf will automatically uncomment that code when processing the template:

```
var x = 23;

/*[+

var msg  = 'This is a working application';

+]*/

var f = function() {
    ...
```

Will be executed as:

```
var x = 23;

var msg  = 'This is a working application';

var f = function() {
...
```

You can include expressions inside these comments, and they will be evaluated:

```
var x = 23;

/*[+

var msg  = 'Hello, ' + [[${session.user.name}]];

+]*/

var f = function() {
...
```

Removing code

It is also possible to make Thymeleaf remove code between special `/*[- */` and `/* -]*/` comments, like this:

```
var x = 23;
```

```
/*[- */

var msg  = 'This is a non-working template';

/* -]*/

var f = function() {
...
```

13 Validation and Doctypes

13.1 Validating templates

As mentioned before, Thymeleaf offers us out-of-the-box two standard template modes that validate our templates before processing them: `VALIDXML` and `VALIDXHTML`. These modes require our templates to be not only *well-formed XML* (which they should always be), but in fact valid according to the specified `DTD`.

The problem is that if we use the `VALIDXHTML` mode with templates including a `DOCTYPE` clause such as this:

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

...we are going to obtain validation errors because the `th:*` tags do not exist according to that `DTD`. That's perfectly normal, as the W3C obviously has no reason to include Thymeleaf's features in their standards but, how do we solve it? By changing the `DTD`.

Thymeleaf includes a set of `DTD` files that mirror the original ones from the XHTML standards, but adding all the available `th:*` attributes from the Standard Dialect. That's why we have been using this in our templates:

```
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-4.dtd">
```

That `SYSTEM` identifier instructs the Thymeleaf parser to resolve the special Thymeleaf-enabled `XHTML 1.0 Strict DTD` file and use it for validating our template. And don't worry about that `http` thing, because that is only an identifier, and the `DTD` file will be locally read from Thymeleaf's jar files.

> Note that because this DOCTYPE declaration is a perfectly valid one, if we open a browser to statically display our template as a prototype it will be rendered in *Standards Mode*.

Here you have the complete set of Thymeleaf-enabled `DTD` declarations for all the supported flavours of XHTML:

```
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-4.dtd">
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-transitional-thymeleaf-4.dtd">
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-frameset-thymeleaf-4.dtd">
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml11-thymeleaf-4.dtd">
```

Also note that, in order for your IDE to be happy, and even if you are not working in a validating mode, you will need to declare the `th` namespace in your `html` tag:

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

13.2 Doctype translation

It is fine for our templates to have a `DOCTYPE` like:

```
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-4.dtd">
```

But it would not be fine for our web applications to send XHTML documents with this `DOCTYPE` to client browsers, because:

- They are not `PUBLIC` (they are `SYSTEM DOCTYPE`s), and therefore our web would not be validatable with the

W3C Validators.

- They are not needed, because once processed, all `th:*` tags will have dissapeared.

That's why Thymeleaf includes a mechanism for *DOCTYPE translation*, which will automatically translate your thymeleaf-specific XHTML `DOCTYPE`s into standard `DOCTYPE`s.

For example, if your template is *XHTML 1.0 Strict* and looks like this:

```
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-4.dtd">

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

After making Thymeleaf process the template, your resulting XHTML will look like this:

```
<!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">
    ...
</html>
```

You don't have to do anything for these transformations to take place: Thymeleaf will take care of them automatically.

14 Some more Pages for our Grocery

Now we know a lot about using Thymeleaf, we can add some new pages to our website for order management.

Note that we will focus on XHTML code, but you can have a look at the bundled source code if you want to see the corresponding controllers.

14.1 Order List

Let's start by creating an order list page, `/WEB-INF/templates/order/list.html`:

```
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-4.dtd">

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

  <head>

    <title>Good Thymes Virtual Grocery</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <link rel="stylesheet" type="text/css" media="all"
          href="../../../css/gtvg.css" th:href="@{/css/gtvg.css}" />
  </head>

  <body>

    <h1>Order list</h1>

    <table>
      <tr>
        <th>DATE</th>
        <th>CUSTOMER</th>
        <th>TOTAL</th>
        <th></th>
      </tr>
      <tr th:each="o : ${orders}" th:class="${oStat.odd}? 'odd'">
        <td th:text="${#calendars.format(o.date,'dd/MMM/yyyy')}">13 jan 2011</td>
        <td th:text="${o.customer.name}">Frederic Tomato</td>
        <td th:text="${#aggregates.sum(o.orderLines.{purchasePrice * amount})}">23.32</td>
        <td>
          <a href="details.html" th:href="@{/order/details(orderId=${o.id})}">view</a>
        </td>
      </tr>
    </table>

    <p>
      <a href="../home.html" th:href="@{/}">Return to home</a>
    </p>

  </body>

</html>
```

There's nothing here that should surprise us, except for this little bit of OGNL magic:

```
<td th:text="${#aggregates.sum(o.orderLines.{purchasePrice * amount})}">23.32</td>
```

What that does is, for each order line (`OrderLine` object) in the order, multiply its `purchasePrice` and `amount` properties (by calling the corresponding `getPurchasePrice()` and `getAmount()` methods) and return the result into a list of numbers, later aggregated by the `#aggregates.sum(...)` function in order to obtain the order total price.

You've got to love the power of OGNL.

14.2 Order Details

Now for the order details page, in which we will make a heavy use of asterisk syntax:

```
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-4.dtd">

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

  <head>
    <title>Good Thymes Virtual Grocery</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <link rel="stylesheet" type="text/css" media="all"
          href="../../../css/gtvg.css" th:href="@{/css/gtvg.css}" />
  </head>

  <body th:object="${order}">

    <h1>Order details</h1>

    <div>
      <p><b>Code:</b> <span th:text="*{id}">99</span></p>
      <p>
        <b>Date:</b>
        <span th:text="*{#calendars.format(date,'dd MMM yyyy')}">13 jan 2011</span>
      </p>
    </div>

    <h2>Customer</h2>

    <div th:object="*{customer}">
      <p><b>Name:</b> <span th:text="*{name}">Frederic Tomato</span></p>
      <p>
        <b>Since:</b>
        <span th:text="*{#calendars.format(customerSince,'dd MMM yyyy')}">1 jan 2011</span>
      </p>
    </div>

    <h2>Products</h2>

    <table>
      <tr>
        <th>PRODUCT</th>
        <th>AMOUNT</th>
        <th>PURCHASE PRICE</th>
      </tr>
      <tr th:each="ol,row : *{orderLines}" th:class="${row.odd}? 'odd'">
        <td th:text="${ol.product.name}">Strawberries</td>
        <td th:text="${ol.amount}" class="number">3</td>
        <td th:text="${ol.purchasePrice}" class="number">23.32</td>
      </tr>
    </table>

    <div>
      <b>TOTAL:</b>
      <span th:text="*{#aggregates.sum(orderLines.{purchasePrice * amount})}">35.23</span>
    </div>

    <p>
      <a href="list.html" th:href="@{/order/list}">Return to order list</a>
    </p>

  </body>

</html>
```

Not much really new here, except for this nested object selection:

```
<body th:object="${order}">

  ...

  <div th:object="*{customer}">
```

```
    <p><b>Name:</b> <span th:text="*{name}">Frederic Tomato</span></p>
    ...
  </div>

  ...
</body>
```

...which makes that `*{name}` in fact equivalent to:

```
<p><b>Name:</b> <span th:text="${order.customer.name}">Frederic Tomato</span></p>
```

15 More on Configuration

15.1 Template Resolvers

For our Good Thymes Virtual Grocery, we chose an `ITemplateResolver` implementation called `ServletContextTemplateResolver` that allowed us to obtain templates as resources from the Servlet Context.

Besides giving you the ability to create your own template resolver by implementing `ITemplateResolver`, Thymeleaf includes three other implementations out of the box:

- `org.thymeleaf.templateresolver.ClassLoaderTemplateResolver`, which resolves templates as classloader resources, like:

```
return Thread.currentThread().getContextClassLoader().getResourceAsStream(templateName);
```

- `org.thymeleaf.templateresolver.FileTemplateResolver`, which resolves templates as files from the file system, like:

```
return new FileInputStream(new File(templateName));
```

- `org.thymeleaf.templateresolver.UrlTemplateResolver`, which resolves templates as URLs (even non-local ones), like:

```
return (new URL(templateName)).openStream();
```

All of the pre-bundled implementations of `ITemplateResolver` allow the same set of configuration parameters, which include:

- Prefix and suffix (as already seen):

```
templateResolver.setPrefix("/WEB-INF/templates/");
templateResolver.setSuffix(".html");
```

- Template aliases that allow the use of template names that do not directly correspond to file names. If both suffix/prefix and alias exist, alias will be applied before prefix/suffix:

```
templateResolver.addTemplateAlias("adminHome","profiles/admin/home");
templateResolver.setTemplateAliases(aliasesMap);
```

- Encoding to be applied when reading templates:

```
templateResolver.setEncoding("UTF-8");
```

- Default template mode, and patterns for defining other modes for specific templates:

```
// Default is TemplateMode.XHTML
templateResolver.setTemplateMode("HTML5");
templateResolver.getXhtmlTemplateModePatternSpec().addPattern("*.xhtml");
```

- Default mode for template cache, and patterns for defining whether specific templates are cacheable or not:

```
// Default is true
templateResolver.setCacheable(false);
templateResolver.getCacheablePatternSpec().addPattern("/users/*");
```

- TTL in milliseconds for parsed template cache entries originated in this template resolver. If not set, the only way to remove an entry from the cache will be LRU (cache max size exceeded and the entry is the oldest).

```
// Default is no TTL (only LRU would remove entries)
templateResolver.setCacheTTLMs(60000L);
```

Also, a Template Engine can be specified several template resolvers, in which case an order can be established between them for template resolution so that, if the first one is not able to resolve the template, the second one is asked, and so on:

```
ClassLoaderTemplateResolver classLoaderTemplateResolver = new ClassLoaderTemplateResolver();
classLoaderTemplateResolver.setOrder(Integer.valueOf(1));

ServletContextTemplateResolver servletContextTemplateResolver = new
ServletContextTemplateResolver();
servletContextTemplateResolver.setOrder(Integer.valueOf(2));

templateEngine.addTemplateResolver(classLoaderTemplateResolver);
templateEngine.addTemplateResolver(servletContextTemplateResolver);
```

When several template resolvers are applied, it is recommended to specify patterns for each template resolver so that Thymeleaf can quickly discard those template resolvers that are not meant to resolve the template, enhancing performance. Doing this is not a requirement, but an optimization:

```
ClassLoaderTemplateResolver classLoaderTemplateResolver = new ClassLoaderTemplateResolver();
classLoaderTemplateResolver.setOrder(Integer.valueOf(1));
// This classloader will not be even asked for any templates not matching these patterns
classLoaderTemplateResolver.getResolvablePatternSpec().addPattern("/layout/*.html");
classLoaderTemplateResolver.getResolvablePatternSpec().addPattern("/menu/*.html");

ServletContextTemplateResolver servletContextTemplateResolver = new
ServletContextTemplateResolver();
servletContextTemplateResolver.setOrder(Integer.valueOf(2));
```

15.2 Message Resolvers

We did not explicitly specify a Message Resolver implementation for our Grocery application, and as it was explained before, this meant that the implementation being used was an `org.thymeleaf.messageresolver.StandardMessageResolver` object.

This `StandardMessageResolver`, which looks for messages files with the same name as the template in the way already explained, is in fact the only message resolver implementation offered by Thymeleaf core out of the box, although of course you can create your own by just implementing the `org.thymeleaf.messageresolver.IMessageResolver` interface.

> The Thymeleaf + Spring integration packages offer an `IMessageResolver` implementation which uses the standard Spring way of retrieving externalized messages, by using `MessageSource` objects.

What if you wanted to add a message resolver (or more) to the Template Engine? Easy:

```
// For setting only one
templateEngine.setMessageResolver(messageResolver);

// For setting more than one
templateEngine.addMessageResolver(messageResolver);
```

And why would you want to have more than one message resolver? for the same reason as template resolvers: message resolvers are ordered and if the first one cannot resolve a specific message, the second one will be asked, then the third, etc.

15.3 Logging

Thymeleaf pays quite a lot of attention to logging, and always tries to offer the maximum amount of useful information through its logging interface.

The logging library used is `slf4j`, which in fact acts as a bridge to whichever logging implementation you might want to use in your application (for example, `log4j`).

Thymeleaf classes will log `TRACE`, `DEBUG` and `INFO`-level information, depending on the level of detail you desire, and besides general logging it will use three special loggers associated with the TemplateEngine class which you can configure separately for different purposes:

- `org.thymeleaf.TemplateEngine.CONFIG` will output detailed configuration of the library during initialization.
- `org.thymeleaf.TemplateEngine.TIMER` will output information about the amount of time taken to process each template (useful for benchmarking!)
- `org.thymeleaf.TemplateEngine.cache` is the prefix for a set of loggers that output specific information about the caches. Although the names of the cache loggers are configurable by the user and thus could change, by default they are:
    - `org.thymeleaf.TemplateEngine.cache.TEMPLATE_CACHE`
    - `org.thymeleaf.TemplateEngine.cache.FRAGMENT_CACHE`
    - `org.thymeleaf.TemplateEngine.cache.MESSAGE_CACHE`
    - `org.thymeleaf.TemplateEngine.cache.EXPRESSION_CACHE`

An example configuration for Thymeleaf's logging infrastructure, using `log4j`, could be:

```
log4j.logger.org.thymeleaf=DEBUG
log4j.logger.org.thymeleaf.TemplateEngine.CONFIG=TRACE
log4j.logger.org.thymeleaf.TemplateEngine.TIMER=TRACE
log4j.logger.org.thymeleaf.TemplateEngine.cache.TEMPLATE_CACHE=TRACE
```

16 Template Cache

Thymeleaf works thanks to a DOM processing engine and a series of processors —one for each type of node that needs to apply logic— that modify the document's DOM tree in order to create the results you expect by combining this tree with your data.

It also includes —by default— a cache that stores parsed templates, this is, the DOM trees resulting from reading and parsing template files before processing them. This is especially useful when working in a web application, and builds on the following concepts:

- Input/Output is almost always the slowest part of any application. In-memory process is extremely quick compared to it.
- Cloning an existing in-memory DOM-tree is always much quicker than reading a template file, parsing it and creating a new DOM object tree for it.
- Web applications usually only have a few dozen templates.
- Template files are small-to-medium size, and they are not modified while the application is running.

This all leads to the idea that caching the most used templates in a web application is feasible without wasting big amounts of memory, and also that it will save a lot of time that would be spent on input/output operations on a small set of files that, in fact, never change.

And how can we take control of this cache? First, we've learned before that we can enable or disable it at the Template Resolver, even acting only on specific templates:

```
// Default is true
templateResolver.setCacheable(false);
templateResolver.getCacheablePatternSpec().addPattern("/users/*");
```

Also, we could modify its configuration by establishing our own *Cache Manager* object, which could be an instance of the default `StandardCacheManager` implementation:

```
// Default is 50
StandardCacheManager cacheManager = new StandardCacheManager();
cacheManager.setTemplateCacheMaxSize(100);
...
templateEngine.setCacheManager(cacheManager);
```

Refer to the javadoc API of `org.thymeleaf.cache.StandardCacheManager` for more info on configuring the caches.

Entries can be manually removed from the template cache:

```
// Clear the cache completely
templateEngine.clearTemplateCache();

// Clear a specific template from the cache
templateEngine.clearTemplateCacheFor("/users/userList");
```

17 Appendix A: Expression Basic Objects

Some objects and variable maps are always available to be invoked at variable expressions (executed by OGNL or SpringEL). Let's see them:

Base objects

- **`#ctx`** : the context object. It will be an implementation of `org.thymeleaf.context.IContext`, `org.thymeleaf.context.IWebContext` depending on our environment (standalone or web). If we are using the *Spring integration module*, it will be an instance of `org.thymeleaf.spring[3|4].context.SpringWebContext`.

```
/*
 * ======================================================================
 * See javadoc API for class org.thymeleaf.context.IContext
 * ======================================================================
 */

${#ctx.locale}
${#ctx.variables}

/*
 * ======================================================================
 * See javadoc API for class org.thymeleaf.context.IWebContext
 * ======================================================================
 */

${#ctx.applicationAttributes}
${#ctx.httpServletRequest}
${#ctx.httpServletResponse}
${#ctx.httpSession}
${#ctx.requestAttributes}
${#ctx.requestParameters}
${#ctx.servletContext}
${#ctx.sessionAttributes}
```

- **`#locale`** : direct access to the `java.util.Locale` associated with current request.

```
${#locale}
```

- **`#vars`** : an instance of `org.thymeleaf.context.VariablesMap` with all the variables in the Context (usually the variables contained in `#ctx.variables` plus local ones).

  Unqualified expressions are evaluated against this object. In fact, `${something}` is completely equivalent to (but more beautiful than) `${#vars.something}`.

  `#root` is a synomyn for the same object.

```
/*
 * ======================================================================
 * See javadoc API for class org.thymeleaf.context.VariablesMap
 * ======================================================================
 */

${#vars.get('foo')}
${#vars.containsKey('foo')}
${#vars.size()}
...
```

Web context namespaces for request/session attributes, etc.

When using Thymeleaf in a web environment, we can use a series of shortcuts for accessing request parameters,

session attributes and application attributes:

> Note these are not *context objects*, but maps added to the context as variables, so we access them without `#`. In some way, therefore, they act as *namespaces*.

- **param** : for retrieving request parameters. `${param.foo}` is a `String[]` with the values of the `foo` request parameter, so `${param.foo[0]}` will normally be used for getting the first value.

```
/*
 * ======================================================================
 * See javadoc API for class org.thymeleaf.context.WebRequestParamsVariablesMap
 * ======================================================================
 */

${param.foo}              // Retrieves a String[] with the values of request parameter 'foo'
${param.size()}
${param.isEmpty()}
${param.containsKey('foo')}
...
```

- **session** : for retrieving session attributes.

```
/*
 * ======================================================================
 * See javadoc API for class org.thymeleaf.context.WebSessionVariablesMap
 * ======================================================================
 */

${session.foo}                // Retrieves the session atttribute 'foo'
${session.size()}
${session.isEmpty()}
${session.containsKey('foo')}
...
```

- **application** : for retrieving application/servlet context attributes.

```
/*
 * ======================================================================
 * See javadoc API for class org.thymeleaf.context.WebServletContextVariablesMap
 * ======================================================================
 */

${application.foo}             // Retrieves the ServletContext atttribute 'foo'
${application.size()}
${application.isEmpty()}
${application.containsKey('foo')}
...
```

Note there is **no need to specify a namespace for accessing request attributes** (as opposed to *request parameters*) because all request attributes are automatically added to the context as variables in the context root:

```
${myRequestAttribute}
```

Web context objects

Inside a web environment there is also direct access to the following objects (note these are objects, not maps/namespaces):

- **#httpServletRequest** : direct access to the `javax.servlet.http.HttpServletRequest` object associated with the current request.

```
${#httpServletRequest.getAttribute('foo')}
${#httpServletRequest.getParameter('foo')}
```

```
${#httpServletRequest.getContextPath()}
${#httpServletRequest.getRequestName()}
...
```

- **#httpSession** : direct access to the `javax.servlet.http.HttpSession` object associated with the current request.

```
${#httpSession.getAttribute('foo')}
${#httpSession.id}
${#httpSession.lastAccessedTime}
...
```

Spring context objects

If you are using Thymeleaf from Spring, you can also access these objects:

- **#themes** : provides the same features as the Spring `spring:theme` JSP tag.

```
${#themes.code('foo')}
```

Spring beans

Thymeleaf also allows accessing beans registered at your Spring Application Context in the standard way defined by Spring EL, which is using the syntax `@beanName`, for example:

```
<div th:text="${@authService.getUserName()}">...</div>
```

18 Appendix B: Expression Utility Objects

Dates

- **`#dates`** : utility methods for `java.util.Date` objects:

```
/*
 * =====================================================================
 * See javadoc API for class org.thymeleaf.expression.Dates
 * =====================================================================
 */

/*
 * Format date with the standard locale format
 * Also works with arrays, lists or sets
 */
${#dates.format(date)}
${#dates.arrayFormat(datesArray)}
${#dates.listFormat(datesList)}
${#dates.setFormat(datesSet)}

/*
 * Format date with the specified pattern
 * Also works with arrays, lists or sets
 */
${#dates.format(date, 'dd/MMM/yyyy HH:mm')}
${#dates.arrayFormat(datesArray, 'dd/MMM/yyyy HH:mm')}
${#dates.listFormat(datesList, 'dd/MMM/yyyy HH:mm')}
${#dates.setFormat(datesSet, 'dd/MMM/yyyy HH:mm')}

/*
 * Obtain date properties
 * Also works with arrays, lists or sets
 */
${#dates.day(date)}                     // also arrayDay(...), listDay(...), etc.
${#dates.month(date)}                   // also arrayMonth(...), listMonth(...), etc.
${#dates.monthName(date)}               // also arrayMonthName(...), listMonthName(...), etc.
${#dates.monthNameShort(date)}          // also arrayMonthNameShort(...),
listMonthNameShort(...), etc.
${#dates.year(date)}                    // also arrayYear(...), listYear(...), etc.
${#dates.dayOfWeek(date)}               // also arrayDayOfWeek(...), listDayOfWeek(...), etc.
${#dates.dayOfWeekName(date)}           // also arrayDayOfWeekName(...),
listDayOfWeekName(...), etc.
${#dates.dayOfWeekNameShort(date)}      // also arrayDayOfWeekNameShort(...),
listDayOfWeekNameShort(...), etc.
${#dates.hour(date)}                    // also arrayHour(...), listHour(...), etc.
${#dates.minute(date)}                  // also arrayMinute(...), listMinute(...), etc.
${#dates.second(date)}                  // also arraySecond(...), listSecond(...), etc.
${#dates.millisecond(date)}             // also arrayMillisecond(...), listMillisecond(...),
etc.

/*
 * Create date (java.util.Date) objects from its components
 */
${#dates.create(year,month,day)}
${#dates.create(year,month,day,hour,minute)}
${#dates.create(year,month,day,hour,minute,second)}
${#dates.create(year,month,day,hour,minute,second,millisecond)}

/*
 * Create a date (java.util.Date) object for the current date and time
 */
${#dates.createNow()}

/*
 * Create a date (java.util.Date) object for the current date (time set to 00:00)
 */
${#dates.createToday()}
```

Calendars

- **#calendars** : analogous to `#dates`, but for `java.util.Calendar` objects:

```
/*
 * ======================================================================
 * See javadoc API for class org.thymeleaf.expression.Calendars
 * ======================================================================
 */

/*
 * Format calendar with the standard locale format
 * Also works with arrays, lists or sets
 */
${#calendars.format(cal)}
${#calendars.arrayFormat(calArray)}
${#calendars.listFormat(calList)}
${#calendars.setFormat(calSet)}

/*
 * Format calendar with the specified pattern
 * Also works with arrays, lists or sets
 */
${#calendars.format(cal, 'dd/MMM/yyyy HH:mm')}
${#calendars.arrayFormat(calArray, 'dd/MMM/yyyy HH:mm')}
${#calendars.listFormat(calList, 'dd/MMM/yyyy HH:mm')}
${#calendars.setFormat(calSet, 'dd/MMM/yyyy HH:mm')}

/*
 * Obtain calendar properties
 * Also works with arrays, lists or sets
 */
${#calendars.day(date)}                // also arrayDay(...), listDay(...), etc.
${#calendars.month(date)}              // also arrayMonth(...), listMonth(...), etc.
${#calendars.monthName(date)}          // also arrayMonthName(...), listMonthName(...), etc.
${#calendars.monthNameShort(date)}     // also arrayMonthNameShort(...),
listMonthNameShort(...), etc.
${#calendars.year(date)}               // also arrayYear(...), listYear(...), etc.
${#calendars.dayOfWeek(date)}          // also arrayDayOfWeek(...), listDayOfWeek(...), etc.
${#calendars.dayOfWeekName(date)}      // also arrayDayOfWeekName(...),
listDayOfWeekName(...), etc.
${#calendars.dayOfWeekNameShort(date)} // also arrayDayOfWeekNameShort(...),
listDayOfWeekNameShort(...), etc.
${#calendars.hour(date)}               // also arrayHour(...), listHour(...), etc.
${#calendars.minute(date)}             // also arrayMinute(...), listMinute(...), etc.
${#calendars.second(date)}             // also arraySecond(...), listSecond(...), etc.
${#calendars.millisecond(date)}        // also arrayMillisecond(...), listMillisecond(...),
etc.

/*
 * Create calendar (java.util.Calendar) objects from its components
 */
${#calendars.create(year,month,day)}
${#calendars.create(year,month,day,hour,minute)}
${#calendars.create(year,month,day,hour,minute,second)}
${#calendars.create(year,month,day,hour,minute,second,millisecond)}

/*
 * Create a calendar (java.util.Calendar) object for the current date and time
 */
${#calendars.createNow()}

/*
 * Create a calendar (java.util.Calendar) object for the current date (time set to 00:00)
 */
${#calendars.createToday()}
```

Numbers

- **#numbers** : utility methods for number objects:

```
/*
```

```
 * ======================================================================
 * See javadoc API for class org.thymeleaf.expression.Numbers
 * ======================================================================
 */

/*
 * ==========================
 * Formatting integer numbers
 * ==========================
 */

/*
 * Set minimum integer digits.
 * Also works with arrays, lists or sets
 */
${#numbers.formatInteger(num,3)}
${#numbers.arrayFormatInteger(numArray,3)}
${#numbers.listFormatInteger(numList,3)}
${#numbers.setFormatInteger(numSet,3)}


/*
 * Set minimum integer digits and thousands separator:
 * 'POINT', 'COMMA', 'NONE' or 'DEFAULT' (by locale).
 * Also works with arrays, lists or sets
 */
${#numbers.formatInteger(num,3,'POINT')}
${#numbers.arrayFormatInteger(numArray,3,'POINT')}
${#numbers.listFormatInteger(numList,3,'POINT')}
${#numbers.setFormatInteger(numSet,3,'POINT')}


/*
 * ==========================
 * Formatting decimal numbers
 * ==========================
 */

/*
 * Set minimum integer digits and (exact) decimal digits.
 * Also works with arrays, lists or sets
 */
${#numbers.formatDecimal(num,3,2)}
${#numbers.arrayFormatDecimal(numArray,3,2)}
${#numbers.listFormatDecimal(numList,3,2)}
${#numbers.setFormatDecimal(numSet,3,2)}

/*
 * Set minimum integer digits and (exact) decimal digits, and also decimal separator.
 * Also works with arrays, lists or sets
 */
${#numbers.formatDecimal(num,3,2,'COMMA')}
${#numbers.arrayFormatDecimal(numArray,3,2,'COMMA')}
${#numbers.listFormatDecimal(numList,3,2,'COMMA')}
${#numbers.setFormatDecimal(numSet,3,2,'COMMA')}

/*
 * Set minimum integer digits and (exact) decimal digits, and also thousands and
 * decimal separator.
 * Also works with arrays, lists or sets
 */
${#numbers.formatDecimal(num,3,'POINT',2,'COMMA')}
${#numbers.arrayFormatDecimal(numArray,3,'POINT',2,'COMMA')}
${#numbers.listFormatDecimal(numList,3,'POINT',2,'COMMA')}
${#numbers.setFormatDecimal(numSet,3,'POINT',2,'COMMA')}



/*
 * ==========================
 * Utility methods
 * ==========================
 */

/*
 * Create a sequence (array) of integer numbers going
 * from x to y
```

```
 */
${#numbers.sequence(from,to)}
${#numbers.sequence(from,to,step)}
```

Strings

- **`#strings`** : utility methods for `String` objects:

```
/*
 * ========================================================================
 * See javadoc API for class org.thymeleaf.expression.Strings
 * ========================================================================
 */

/*
 * Null-safe toString()
 */
${#strings.toString(obj)}                           // also array*, list* and set*

/*
 * Check whether a String is empty (or null). Performs a trim() operation before check
 * Also works with arrays, lists or sets
 */
${#strings.isEmpty(name)}
${#strings.arrayIsEmpty(nameArr)}
${#strings.listIsEmpty(nameList)}
${#strings.setIsEmpty(nameSet)}

/*
 * Perform an 'isEmpty()' check on a string and return it if false, defaulting to
 * another specified string if true.
 * Also works with arrays, lists or sets
 */
${#strings.defaultString(text,default)}
${#strings.arrayDefaultString(textArr,default)}
${#strings.listDefaultString(textList,default)}
${#strings.setDefaultString(textSet,default)}

/*
 * Check whether a fragment is contained in a String
 * Also works with arrays, lists or sets
 */
${#strings.contains(name,'ez')}                     // also array*, list* and set*
${#strings.containsIgnoreCase(name,'ez')}           // also array*, list* and set*

/*
 * Check whether a String starts or ends with a fragment
 * Also works with arrays, lists or sets
 */
${#strings.startsWith(name,'Don')}                  // also array*, list* and set*
${#strings.endsWith(name,endingFragment)}           // also array*, list* and set*

/*
 * Substring-related operations
 * Also works with arrays, lists or sets
 */
${#strings.indexOf(name,frag)}                      // also array*, list* and set*
${#strings.substring(name,3,5)}                     // also array*, list* and set*
${#strings.substringAfter(name,prefix)}             // also array*, list* and set*
${#strings.substringBefore(name,suffix)}            // also array*, list* and set*
${#strings.replace(name,'las','ler')}               // also array*, list* and set*

/*
 * Append and prepend
 * Also works with arrays, lists or sets
 */
${#strings.prepend(str,prefix)}                     // also array*, list* and set*
${#strings.append(str,suffix)}                      // also array*, list* and set*

/*
 * Change case
 * Also works with arrays, lists or sets
 */
${#strings.toUpperCase(name)}                       // also array*, list* and set*
```

```
${#strings.toLowerCase(name)}                       // also array*, list* and set*

/*
 * Split and join
 */
${#strings.arrayJoin(namesArray,',')}
${#strings.listJoin(namesList,',')}
${#strings.setJoin(namesSet,',')}
${#strings.arraySplit(namesStr,',')}                // returns String[]
${#strings.listSplit(namesStr,',')}                 // returns List<String>
${#strings.setSplit(namesStr,',')}                  // returns Set<String>

/*
 * Trim
 * Also works with arrays, lists or sets
 */
${#strings.trim(str)}                               // also array*, list* and set*

/*
 * Compute length
 * Also works with arrays, lists or sets
 */
${#strings.length(str)}                             // also array*, list* and set*

/*
 * Abbreviate text making it have a maximum size of n. If text is bigger, it
 * will be clipped and finished in "..."
 * Also works with arrays, lists or sets
 */
${#strings.abbreviate(str,10)}                      // also array*, list* and set*

/*
 * Convert the first character to upper-case (and vice-versa)
 */
${#strings.capitalize(str)}                         // also array*, list* and set*
${#strings.unCapitalize(str)}                       // also array*, list* and set*

/*
 * Convert the first character of every word to upper-case
 */
${#strings.capitalizeWords(str)}                    // also array*, list* and set*
${#strings.capitalizeWords(str,delimiters)}         // also array*, list* and set*

/*
 * Escape the string
 */
${#strings.escapeXml(str)}                          // also array*, list* and set*
${#strings.escapeJava(str)}                         // also array*, list* and set*
${#strings.escapeJavaScript(str)}                   // also array*, list* and set*
${#strings.unescapeJava(str)}                       // also array*, list* and set*
${#strings.unescapeJavaScript(str)}                 // also array*, list* and set*

/*
 * Null-safe comparison and concatenation
 */
${#strings.equals(str)}
${#strings.equalsIgnoreCase(str)}
${#strings.concat(str)}
${#strings.concatReplaceNulls(str)}

/*
 * Random
 */
${#strings.randomAlphanumeric(count)}
```

Objects

- **#objects** : utility methods for objects in general

```
/*
 * ========================================================================
 * See javadoc API for class org.thymeleaf.expression.Objects
 * ========================================================================
 */
```

```
/*
 * Return obj if it is not null, and default otherwise
 * Also works with arrays, lists or sets
 */
${#objects.nullSafe(obj,default)}
${#objects.arrayNullSafe(objArray,default)}
${#objects.listNullSafe(objList,default)}
${#objects.setNullSafe(objSet,default)}
```

Booleans

- **#bools** : utility methods for boolean evaluation

```
/*
 * =============================================================================
 * See javadoc API for class org.thymeleaf.expression.Bools
 * =============================================================================
 */

/*
 * Evaluate a condition in the same way that it would be evaluated in a th:if tag
 * (see conditional evaluation chapter afterwards).
 * Also works with arrays, lists or sets
 */
${#bools.isTrue(obj)}
${#bools.arrayIsTrue(objArray)}
${#bools.listIsTrue(objList)}
${#bools.setIsTrue(objSet)}

/*
 * Evaluate with negation
 * Also works with arrays, lists or sets
 */
${#bools.isFalse(cond)}
${#bools.arrayIsFalse(condArray)}
${#bools.listIsFalse(condList)}
${#bools.setIsFalse(condSet)}

/*
 * Evaluate and apply AND operator
 * Receive an array, a list or a set as parameter
 */
${#bools.arrayAnd(condArray)}
${#bools.listAnd(condList)}
${#bools.setAnd(condSet)}

/*
 * Evaluate and apply OR operator
 * Receive an array, a list or a set as parameter
 */
${#bools.arrayOr(condArray)}
${#bools.listOr(condList)}
${#bools.setOr(condSet)}
```

Arrays

- **#arrays** : utility methods for arrays

```
/*
 * =============================================================================
 * See javadoc API for class org.thymeleaf.expression.Arrays
 * =============================================================================
 */

/*
 * Converts to array, trying to infer array component class.
 * Note that if resulting array is empty, or if the elements
 * of the target object are not all of the same class,
 * this method will return Object[].
```

```
 */
${#arrays.toArray(object)}

/*
 * Convert to arrays of the specified component class.
 */
${#arrays.toStringArray(object)}
${#arrays.toIntegerArray(object)}
${#arrays.toLongArray(object)}
${#arrays.toDoubleArray(object)}
${#arrays.toFloatArray(object)}
${#arrays.toBooleanArray(object)}

/*
 * Compute length
 */
${#arrays.length(array)}

/*
 * Check whether array is empty
 */
${#arrays.isEmpty(array)}

/*
 * Check if element or elements are contained in array
 */
${#arrays.contains(array, element)}
${#arrays.containsAll(array, elements)}
```

Lists

- **#lists** : utility methods for lists

```
/*
 * ========================================================================
 * See javadoc API for class org.thymeleaf.expression.Lists
 * ========================================================================
 */

/*
 * Converts to list
 */
${#lists.toList(object)}

/*
 * Compute size
 */
${#lists.size(list)}

/*
 * Check whether list is empty
 */
${#lists.isEmpty(list)}

/*
 * Check if element or elements are contained in list
 */
${#lists.contains(list, element)}
${#lists.containsAll(list, elements)}

/*
 * Sort a copy of the given list. The members of the list must implement
 * comparable or you must define a comparator.
 */
${#lists.sort(list)}
${#lists.sort(list, comparator)}
```

Sets

- **#sets** : utility methods for sets

```
/*
 * ============================================================================
 * See javadoc API for class org.thymeleaf.expression.Sets
 * ============================================================================
 */

/*
 * Converts to set
 */
${#sets.toSet(object)}

/*
 * Compute size
 */
${#sets.size(set)}

/*
 * Check whether set is empty
 */
${#sets.isEmpty(set)}

/*
 * Check if element or elements are contained in set
 */
${#sets.contains(set, element)}
${#sets.containsAll(set, elements)}
```

Maps

- **#maps** : utility methods for maps

```
/*
 * ============================================================================
 * See javadoc API for class org.thymeleaf.expression.Maps
 * ============================================================================
 */

/*
 * Compute size
 */
${#maps.size(map)}

/*
 * Check whether map is empty
 */
${#maps.isEmpty(map)}

/*
 * Check if key/s or value/s are contained in maps
 */
${#maps.containsKey(map, key)}
${#maps.containsAllKeys(map, keys)}
${#maps.containsValue(map, value)}
${#maps.containsAllValues(map, value)}
```

Aggregates

- **#aggregates** : utility methods for creating aggregates on arrays or collections

```
/*
 * ============================================================================
 * See javadoc API for class org.thymeleaf.expression.Aggregates
 * ============================================================================
 */

/*
 * Compute sum. Returns null if array or collection is empty
 */
${#aggregates.sum(array)}
${#aggregates.sum(collection)}
```

```
/*
 * Compute average. Returns null if array or collection is empty
 */
${#aggregates.avg(array)}
${#aggregates.avg(collection)}
```

Messages

- **#messages** : utility methods for obtaining externalized messages inside variables expressions, in the same way as they would be obtained using `#{...}` syntax.

```
/*
 * =====================================================================
 * See javadoc API for class org.thymeleaf.expression.Messages
 * =====================================================================
 */

/*
 * Obtain externalized messages. Can receive a single key, a key plus arguments,
 * or an array/list/set of keys (in which case it will return an array/list/set of
 * externalized messages).
 * If a message is not found, a default message (like '??msgKey??') is returned.
 */
${#messages.msg('msgKey')}
${#messages.msg('msgKey', param1)}
${#messages.msg('msgKey', param1, param2)}
${#messages.msg('msgKey', param1, param2, param3)}
${#messages.msgWithParams('msgKey', new Object[] {param1, param2, param3, param4})}
${#messages.arrayMsg(messageKeyArray)}
${#messages.listMsg(messageKeyList)}
${#messages.setMsg(messageKeySet)}

/*
 * Obtain externalized messages or null. Null is returned instead of a default
 * message if a message for the specified key is not found.
 */
${#messages.msgOrNull('msgKey')}
${#messages.msgOrNull('msgKey', param1)}
${#messages.msgOrNull('msgKey', param1, param2)}
${#messages.msgOrNull('msgKey', param1, param2, param3)}
${#messages.msgOrNullWithParams('msgKey', new Object[] {param1, param2, param3, param4})}
${#messages.arrayMsgOrNull(messageKeyArray)}
${#messages.listMsgOrNull(messageKeyList)}
${#messages.setMsgOrNull(messageKeySet)}
```

IDs

- **#ids** : utility methods for dealing with `id` attributes that might be repeated (for example, as a result of an iteration).

```
/*
 * =====================================================================
 * See javadoc API for class org.thymeleaf.expression.Ids
 * =====================================================================
 */

/*
 * Normally used in th:id attributes, for appending a counter to the id attribute value
 * so that it remains unique even when involved in an iteration process.
 */
${#ids.seq('someId')}

/*
 * Normally used in th:for attributes in <label> tags, so that these labels can refer to Ids
 * generated by means if the #ids.seq(...) function.
 *
 * Depending on whether the <label> goes before or after the element with the #ids.seq(...)
 * function, the "next" (label goes before "seq") or the "prev" function (label goes after
 * "seq") function should be called.
```

```
 */
${#ids.next('someId')}
${#ids.prev('someId')}
```

19 Appendix C: DOM Selector syntax

DOM Selectors borrow syntax features from XPATH, CSS and jQuery, in order to provide a powerful and easy to use way to specify template fragments.

For example, the following selector will select every `<div>` with the class `content`, in every position inside the markup:

```
<div th:include="mytemplate :: [//div[@class='content']]">...</div>
```

The basic syntax inspired from XPath includes:

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

But more concise syntax can also be used:

- `x` is exactly equivalent to `//x` (search an element with name or reference `x` at any depth level).
- Selectors are also allowed without element name/reference, as long as they include a specification of arguments. So `[@class='oneclass']` is a valid selector that looks for any elements (tags) with a class attribute with value "oneclass".

Advanced attribute selection features:

- Besides `=` (equal), other comparison operators are also valid: `!=` (not equal), `^=` (starts with) and `$=` (ends with). For example: `x[@class^='section']` means elements with name `x` and a value for attribute `class` that starts with `section`.
- Attributes can be specified both starting with `@` (XPath-style) and without (jQuery-style). So `x[z='v']` is equivalent to `x[@z='v']`.
- Multiple-attribute modifiers can be joined both with `and` (XPath-style) and also by chaining multiple modifiers (jQuery-style). So `x[@z1='v1' and @z2='v2']` is actually equivalent to `x[@z1='v1'][@z2='v2']` (and also to `x[z1='v1'][z2='v2']`).

Direct *jQuery-like* selectors:

- `x.oneclass` is equivalent to `x[class='oneclass']`.
- `.oneclass` is equivalent to `[class='oneclass']`.
- `x#oneid` is equivalent to `x[id='oneid']`.
- `#oneid` is equivalent to `[id='oneid']`.
- `x%oneref` means nodes -not just elements- with name x that match reference *oneref* according to a specified `DOMSelector.INodeReferenceChecker` implementation.

- `%oneref` means nodes -not just elements- with any name that match reference *oneref* according to a specified `DOMSelector.INodeReferenceChecker` implementation. Note this is actually equivalent to simply `oneref` because references can be used instead of element names.
- Direct selectors and attribute selectors can be mixed: `a.external[@href^='https']`.

The above DOM Selector expression:

```
<div th:include="mytemplate :: [//div[@class='content']]">...</div>
```

could be written as:

```
<div th:include="mytemplate :: [div.content]">...</div>
```

Multivalued class matching

DOM Selectors understand the class attribute to be **multivalued**, and therefore allow the application of selectors on this attribute even if the element has several class values.

For example, `div[class='two']` will match `<div class="one two three" />`

Optional brackets

The syntax of the fragment inclusion attributes converts every fragment selection into a DOM selection, so brackets `[...]` are not needed (though allowed).

So the following, with no brackets, is equivalent to the bracketed selector seen above:

```
<div th:include="mytemplate :: div.content">...</div>
```

So, summarizing, this:

```
<div th:replace="mytemplate :: myfrag">...</div>
```

Will look for a `th:fragment="myfrag"` fragment signature. But would also look for tags with name `myfrag` if they existed (which they don't, in HTML). Note the difference with:

```
<div th:replace="mytemplate :: .myfrag">...</div>
```

which will actually look for any elements with `class="myfrag"`, without caring about `th:fragment` signatures.

---

1. Given the fact that XHTML5 is just XML-formed HTML5 served with the application/xhtml+xml content type, we could also say that Thymeleaf supports XHTML5.
2. Note that, although this template is valid XHTML, we earlier selected template mode "XHTML" and not "VALIDXHTML". For now, it will be OK for us to just have validation turned off – but at the same time we don't want our IDE to complain too much.