Thymeleaf

Tutorial: Using Thymeleaf

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

1 Introducing Thymeleaf

1.1 What is Thymeleaf?

Thymeleaf is a modern server-side Java template engine for both web and standalone environments, capable of processing HTML, XML, JavaScript, CSS and even plain text.

The main goal of Thymeleaf is to provide an elegant and highly-maintainable way of creating templates. To achieve this, it builds on the concept of *Natural Templates* to inject its logic into template files in a way that doesn't affect the template from being used as a design prototype. This improves communication of design and bridges the gap between design and development teams.

Thymeleaf has also been designed from the beginning with Web Standards in mind – especially **HTML5** – 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**:

- HTML
- XML
- TEXT
- JAVASCRIPT
- CSS
- RAW

There are two *markup* template modes (`HTML` and `XML`), three *textual* template modes (`TEXT` , `JAVASCRIPT` and `CSS`) and a *no-op* template mode (`RAW`).

The `HTML` template mode will allow any kind of HTML input, including HTML5, HTML 4 and XHTML. No validation or well-formedness check will be performed, and template code/structure will be respected to the biggest possible extent in output.

The `XML` template mode will allow XML input. In this case, code is expected to be well-formed – no unclosed tags, no unquoted attributes, etc – and the parser will throw exceptions if well-formedness violations are found. Note that no *validation* (against a DTD or XML Schema) will be performed.

The `TEXT` template mode will allow the use of a special syntax for templates of a non-markup nature. Examples of such templates might be text emails or templated documentation. Note that HTML or XML templates can be also processed as `TEXT` , in which case they will not be parsed as markup, and every tag, DOCTYPE, comment, etc, will be treated as mere text.

The `JAVASCRIPT` template mode will allow the processing of JavaScript files in a Thymeleaf application. This means being able to use model data inside JavaScript files in the same way it can be done in HTML files, but with JavaScript-specific integrations such as specialized escaping or *natural scripting*. The `JAVASCRIPT` template mode is considered a *textual* mode and therefore uses the same special syntax as the `TEXT` template mode.

The `CSS` template mode will allow the processing of CSS files involved in a Thymeleaf application. Similar to the `JAVASCRIPT` mode, the `CSS` template mode is also a *textual* mode and uses the special processing syntax from the `TEXT` template mode.

The `RAW` template mode will simply not process templates at all. It is meant to be used for inserting untouched resources (files, URL responses, etc.) into the templates being processed. For example, external, uncontrolled resources in HTML

format could be included into application templates, safely knowing that any Thymeleaf code that these resources might include will not be executed.

1.3 Dialects: The Standard Dialect

Thymeleaf is an extremely extensible template engine (in fact it could be called a *template engine framework*) that allows you to define and customize the way your templates will be processed to a fine level of detail.

An object that applies some logic to a markup artifact (a tag, some text, a comment, or a mere placeholder if templates are not markup) is called a *processor*, and a set of these processors – plus perhaps some extra artifacts – is what a **dialect** is normally comprised of. Out of the box, Thymeleaf's core library provides a dialect called the **Standard Dialect**, which should be enough for most users.

> Note that dialects can actually have no processors and be entirely comprised of other kinds of artifacts, but processors are definitely the most common use case.

This tutorial covers the Standard Dialect. 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 can 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. Thymeleaf can also be configured to use several dialects at a time.

> The official thymeleaf-spring3 and thymeleaf-spring4 integration packages both define a dialect called the "SpringStandard Dialect", which is mostly the same as the Standard Dialect, but with small adaptations to make better use of some features in the Spring Framework (for example, by using Spring Expression Language or SpringEL instead of 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.

Most of the processors of the Standard Dialect are *attribute processors*. This allows browsers to correctly display HTML 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}" />
```

Not only will this be correctly displayed by browsers, but this 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 processing of the template.

This helps 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 called *Natural Templating*.

2 The Good Thymes Virtual Grocery

The source code for the examples shown in this, and future chapters of this guide, can be found in the Good Thymes Virtual Grocery GitHub repository.

2.1 A website for a grocery

To better explain the concepts involved in processing templates with Thymeleaf, this tutorial will use a demo application which you can download from the project's web site.

This application is the web site of an imaginary virtual grocery, and will provide us with many scenarios to showcase Thymeleaf's many features.

To start, we need a 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`:
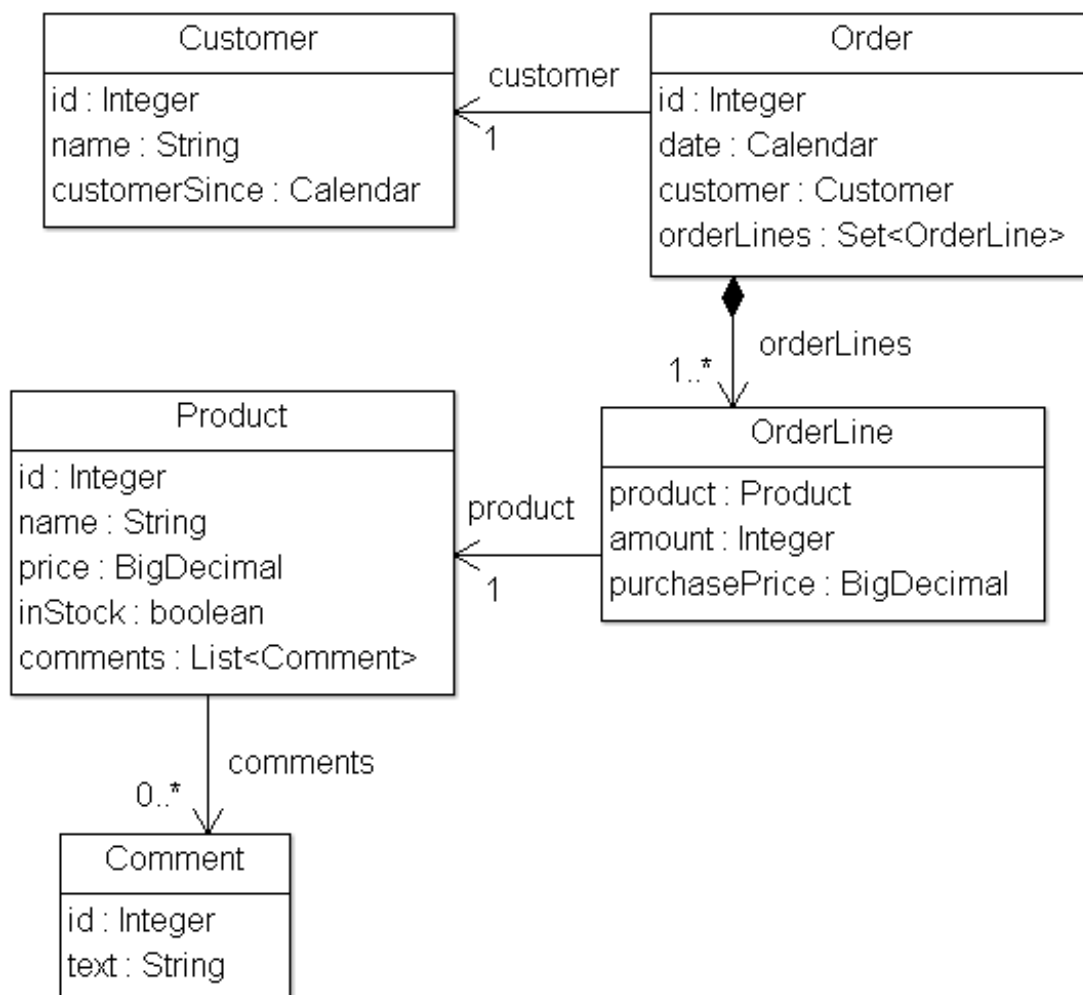
Example application model

Our 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);
    }

}
```

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

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

    try {

        // This prevents triggering engine executions for resource URLs
        if (request.getRequestURI().startsWith("/css") ||
                request.getRequestURI().startsWith("/images") ||
                request.getRequestURI().startsWith("/favicon")) {
            return false;
        }


        /*
         * 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 = this.application.resolveControllerForRequest(request);
        if (controller == null) {
            return false;
        }

        /*
         * Obtain the TemplateEngine instance.
         */
        ITemplateEngine templateEngine = this.application.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) {
        try {
            response.sendError(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
        } catch (final IOException ignored) {
            // Just ignore this
        }
        throw new ServletException(e);
    }

}
```

This is our `IGTVGController` interface:

```
public interface IGTVGController {

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

}
```

All we have to do now is create implementations of the `IGTVGController` interface, retrieving data from the services and processing templates using the `ITemplateEngine` 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 line:

```
ITemplateEngine templateEngine = this.application.getTemplateEngine();
```

Which means that the *GTVGApplication* class is in charge of creating and configuring one of the most important objects in a Thymeleaf application: the `TemplateEngine` instance (implementation of the `ITemplateEngine` interface).

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

```java
public class GTVGApplication {

    ...
    private static TemplateEngine templateEngine;
    ...


    public GTVGApplication(final ServletContext servletContext) {

        super();

        ServletContextTemplateResolver templateResolver =
                new ServletContextTemplateResolver(servletContext);

        // HTML is the default mode, but we set it anyway for better understanding of code
        templateResolver.setTemplateMode(TemplateMode.HTML);
        // 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(Long.valueOf(3600000L));

        // Cache is set to true by default. Set to false if you want templates to
        // be automatically updated when modified.
        templateResolver.setCacheable(true);

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

        ...

    }

}
```

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:

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

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

```java
public interface ITemplateResolver {

    ...

    /*
     * Templates are resolved by their name (or content) and also (optionally) their
     * owner template in case we are trying to resolve a fragment for another template.
     * Will return null if template cannot be handled by this template resolver.
     */
    public TemplateResolution resolveTemplate(
            final IEngineConfiguration configuration,
            final String ownerTemplate, final String template,
            final Map<String, Object> templateResolutionAttributes);
}
```

These objects are in charge of determining how our templates will be accessed, and in this GTVG application, the `org.thymeleaf.templateresolver.ServletContextTemplateResolver` means 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 from the web application root.

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:

```java
templateResolver.setTemplateMode(TemplateMode.HTML);
```

HTML 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.

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

The *prefix* and *suffix* 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:

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

Optionally, the amount of time that a parsed template can live in the cache is configured at the Template Resolver by means of the *cacheTTLMs* property:

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

A template can still 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 by modifying the `StandardCacheManager` object to manage the default cache.

There is much more to learn about template resolvers, but for now let's have a look at the creation of our Template Engine object.

The Template Engine

Template Engine objects are implementations of the `org.thymeleaf.ITemplateEngine` interface. One of these implementations is offered by the Thymeleaf core: `org.thymeleaf.TemplateEngine`, and we create an instance of it here:

```
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 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 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>

<html 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 is that this file is HTML5 that can be correctly displayed by any browser because it does not include any non-HTML tags (browsers ignore all attributes they don't understand, like `th:text`).

But you may also notice that this template is not really a *valid* HTML5 document, because these non-standard attributes we are using in the `th:*` form are not allowed by the HTML5 specification. In fact, we are even adding an `xmlns:th` attribute to our `<html>` tag, something absolutely non-HTML5-ish:

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

...which has no influence at all in template processing, but works as an *incantation* that prevents our IDE from complaining about the lack of a namespace definition for all those `th:*` attributes.

So what if we wanted to make this template **HTML5-valid**? Easy: switch to Thymeleaf's data attribute syntax, using the `data-` prefix for attribute names and hyphen (`-`) separators instead of semi-colons (`:`):

```
<!DOCTYPE html>

<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" data-th-href="@{/css/gtvg.css}" />
  </head>

  <body>

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

  </body>

</html>
```

Custom `data-` prefixed attributes are allowed by the HTML5 specification, so, with this code above, our template would be a *valid HTML5 document*.

> Both notations are completely equivalent and interchangeable, but for the sake of simplicity and compactness of the code samples, this tutorial will use the *namespace notation* (`th:*`). Also, the `th:*` notation is more general and allowed in every Thymeleaf template mode (`XML` , `TEXT` ...) whereas the `data-` notation is only allowed in `HTML` mode.

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 separate files (typically `.properties` files) and that they can be easily replaced with equivalent texts written in other languages (a process called internationalization or simply *i18n*). Externalized fragments of text are usually called *"messages"*.

Messages always have 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 as the body of the host tag, effectively replacing the "Welcome to our grocery store!" text we see in the code.
- The `#{home.welcome}` expression, specified in the *Standard Expression Syntax*, instructing 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 for our template engine during initialization, and that means that our application is using the *Standard Message Resolver*, implemented by `org.thymeleaf.messageresolver.StandardMessageResolver` .

The 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 the 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:

```java
public class HomeController implements IGTVGController {

    public void process(
            final HttpServletRequest request, final HttpServletResponse response,
            final ServletContext servletContext, final ITemplateEngine templateEngine)
            throws Exception {

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

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

    }

}
```

The first thing we see 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.

```java
public interface IContext {

    public Locale getLocale();
    public boolean containsVariable(final String name);
    public Set<String> getVariableNames();
    public Object getVariable(final String name);

}
```

There is a specialized extension of this interface, `org.thymeleaf.context.IWebContext`, meant to be used in ServletAPI-based web applications (like SpringMVC).

```
public interface IWebContext extends IContext {

    public HttpServletRequest getRequest();
    public HttpServletResponse getResponse();
    public HttpSession getSession();
    public ServletContext getServletContext();

}
```

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 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, response, servletContext, request.getLocale());
```

Only three out of those four 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).

There are some specialized expressions that we will be able to use to obtain the request parameters and the request, session and application attributes from the `WebContext` in our templates. For example:

- `${x}` will return a variable `x` stored into the Thymeleaf context or as a *request attribute*.
- `${param.x}` will return a *request parameter* called `x` (which might be multivalued).
- `${session.x}` will return a *session attribute* called `x`.
- `${application.x}` will return a *servlet context attribute* called `x`.

Executing the template engine

With our context object ready, now we can tell the template engine to process the template (by its name) using the context, and passing it a 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>

<html>

  <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 in the browser.

This is the default behaviour of the `th:text` attribute. If we want Thymeleaf to respect our HTML 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 content to our home page. For example, we may 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:

```java
public void process(
        final HttpServletRequest request, final HttpServletResponse response,
        final ServletContext servletContext, final ITemplateEngine templateEngine)
        throws Exception {

    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` variable called `today` to our context, and now we can display it in our template:

```html
<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 replace 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**, and it contains an expression in a language called *OGNL (Object-Graph Navigation Language)* that will be executed on the context variables map we talked about before.

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. The next chapter will show us what all these possibilities are.

4 Standard Expression Syntax

We will take 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 expressions, and more interesting details to learn 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: `@{...}`
  - Fragment 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)`

- Special tokens:

- No-Operation: _

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 them 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 can format to numbers and dates as specified in the API docs for those classes.

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>
```

Several parameters can 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

In Spring MVC-enabled applications OGNL will be replaced with **SpringEL**, but its syntax is very similar to that of OGNL (actually, exactly the same for most common cases).

From OGNL's syntax, we know that the expression in:

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

...is in fact equivalent to this:

```
ctx.getVariable("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>
```

...obtains the user name by executing:

```
((User) ctx.getVariable("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.
- `#request` : (only in Web Contexts) the `HttpServletRequest` object.
- `#response` : (only in Web Contexts) the `HttpServletResponse` object.
- `#session` : (only in Web Contexts) the `HttpSession` object.
- `#servletContext` : (only in Web Contexts) the `ServletContext` 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 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.

- `#execInfo` : information about the template being processed.
- `#messages` : methods for obtaining externalized messages inside variables expressions, in the same way as they would be obtained using #{...} syntax.
- `#uris` : methods for escaping parts of URLs/URIs
- `#conversions` : methods for executing the configured *conversion service* (if any).
- `#dates` : methods for `java.util.Date` objects: formatting, component extraction, etc.
- `#calendars` : analogous to `#dates` , but for `java.util.Calendar` objects.
- `#numbers` : methods for formatting numeric objects.
- `#strings` : methods for `String` objects: contains, startsWith, prepending/appending, etc.
- `#objects` : methods for objects in general.
- `#bools` : methods for boolean evaluation.
- `#arrays` : methods for arrays.
- `#lists` : methods for lists.
- `#sets` : methods for sets.
- `#maps` : methods for maps.
- `#aggregates` : methods for creating aggregates on arrays or collections.
- `#ids` : 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, response, 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)

Not only can variable expressions be written as `${...}`, but also as `*{...}`.

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

And what is a selected object? The result of an expression using the `th:object` attribute. Let's use one 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 also be 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 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: `http://www.thymeleaf.org`
- Relative URLs, which can be:

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

The real processing of these expressions and their conversion to the URLs that will be output is done by implementations of the `org.thymeleaf.linkbuilder.ILinkBuilder` interface that are registered into the `ITemplateEngine` object being used.

By default, a single implementation of this interface is registered of the class `org.thymeleaf.linkbuilder.StandardLinkBuilder`, which is enough for both offline (non-web) and also web scenarios based on the Servlet API. Other scenarios (like integration with non-ServletAPI web frameworks) might need specific implementations of the link builder interface.

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 a modifier attribute: once processed, it will compute the link URL to be used and set that value to the `href` attribute of the `<a>` tag.

- We are allowed to use expressions for URL parameters (as you can see in `orderId=${o.id}`). The required URL-parameter-encoding operations will also be automatically performed.
- If several parameters are needed, these will be separated by commas:
  `@{/order/process(execId=${execId},execType='FAST')}`
- Variable templates are also allowed in URL paths: `@{/order/{orderId}/details(orderId=${orderId})}`
- Relative URLs starting with `/` (eg: `/order/details`) will be automatically prefixed by 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 the 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` attribute allows 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 that we know how to create link URLs, what about adding a small menu in our home page 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>
```

Server root relative URLs

An additional syntax can be used to create server-root-relative (instead of context-root-relative) URLs in order to link to different contexts in the same server. These URLs will be specified like `@{~/path/to/something}`

4.5 Fragments

Fragment expressions are an easy way to represent fragments of markup and move them around templates. This allows us to replicate them, pass them to other templates as arguments, and so on.

The most common use is for fragment insertion using `th:insert` or `th:replace` (more on these in a later section):

```
<div th:insert="~{commons :: main}">...</div>
```

But they can be used anywhere, just as any other variable:

```
<div th:with="frag=~{footer :: #main/text()}">
  <p th:insert="${frag}">
</div>
```

Later in this tutorial there is an entire section devoted to Template Layout, including deeper explanation of fragment

expressions.

4.6 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 using `\'` .

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

Number literals

Numeric literals are just that: numbers.

```html
<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:

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

In this example, the `== false` is written outside the braces, and so it is Thymeleaf that takes care of it. If it were written inside the braces, it would be the responsibility of the OGNL/SpringEL engines:

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

The null literal

The `null` literal can be also used:

```html
<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>
```

4.7 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:

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

4.8 Literal substitutions

Literal substitutions allow for an 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 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}|">
```

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

4.9 Arithmetic operations

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

```
<div 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):

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

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

4.10 Comparators and Equality

Values in expressions can be compared with the `>` , `<` , `>=` and `<=` symbols, and the `==` and `!=` operators can be used to check for 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 `>` .

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

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

4.11 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*, `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>
```

4.12 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, lets you specify two expressions: the first one is used if it doesn't evaluate to null, but if it does then the second one is used.

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>
```

4.13 The No-Operation token

The No-Operation token is represented by an underscore symbol (`_`).

The idea behind this token is to specify that the desired result for an expression is to *do nothing*, i.e. do exactly as if the processable attribute (e.g. `th:text`) was not there at all.

Among other possibilities, this allows developers to use prototyping text as default values. For example, instead of:

```
<span th:text="${user.name} ?: 'no user authenticated'">...</span>
```

...we can directly use *'no user authenticated'* as a prototyping text, which results in code that is both more concise and versatile from a design standpoint:

```
<span th:text="${user.name} ?: _">no user authenticated</span>
```

4.15 Data Conversion / Formatting

Thymeleaf defines a *double-brace* syntax for variable (`${...}`) and selection (`*{...}`) expressions that allows us to apply *data conversion* by means of a configured *conversion service*.

It basically goes like this:

```
<td th:text="${{user.lastAccessDate}}">...</td>
```

Noticed the double brace there?: `${{...}}` . That instructs Thymeleaf to pass the result of the `user.lastAccessDate` expression to the *conversion service* and asks it to perform a **formatting operation** (a conversion to `String`) before writing the result.

Assuming that `user.lastAccessDate` is of type `java.util.Calendar` , if a *conversion service* (implementation of `IStandardConversionService`) has been registered and contains a valid conversion for `Calendar -> String` , it will be applied.

The default implementation of `IStandardConversionService` (the `StandardConversionService` class) simply executes `.toString()` on any object converted to `String` . For more information on how to register a custom *conversion service* implementation, have a look at the More on Configuration section.

> The official thymeleaf-spring3 and thymeleaf-spring4 integration packages transparently integrate Thymeleaf's conversion service mechanism with Spring's own *Conversion Service* infrastructure, so that conversion services and formatters declared in the Spring configuration will be made automatically available to `${{...}}` and `*{{...}}` expressions.

4.14 Preprocessing

In addition to all these features for expression processing, Thymeleaf has the feature of *preprocessing* expressions.

Preprocessing is an execution of the expressions done before the normal one that allows for modification of the expression that will eventually be 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.

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:

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

As with Thymeleaf, this template starts off more like a static prototype than it does 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:

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

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

```html
<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!" 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 `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`).

For example, to set the `value` attribute, use `th:value` :

```
<input type="submit" value="Subscribe!" 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 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:fragment
th:frame	th:frameborder	th:headers
th:height	th:high	th:href
th:hreflang	th:hspace	th:http-equiv

th:icon	th:id	th:inline
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:onabort	th:onafterprint
th:onbeforeprint	th:onbeforeunload	th:onblur
th:oncanplay	th:oncanplaythrough	th:onchange
th:onclick	th:oncontextmenu	th:ondblclick
th:ondrag	th:ondragend	th:ondragenter
th:ondragleave	th:ondragover	th:ondragstart
th:ondrop	th:ondurationchange	th:onemptied
th:onended	th:onerror	th:onfocus
th:onformchange	th:onforminput	th:onhashchange
th:oninput	th:oninvalid	th:onkeydown
th:onkeypress	th:onkeyup	th:onload
th:onloadeddata	th:onloadedmetadata	th:onloadstart
th:onmessage	th:onmousedown	th:onmousemove
th:onmouseout	th:onmouseover	th:onmouseup
th:onmousewheel	th:onoffline	th:ononline
th:onpause	th:onplay	th:onplaying
th:onpopstate	th:onprogress	th:onratechange
th:onreadystatechange	th:onredo	th:onreset
th:onresize	th:onscroll	th:onseeked
th:onseeking	th:onselect	th:onshow
th:onstalled	th:onstorage	th:onsubmit
th:onsuspend	th:ontimeupdate	th:onundo
th:onunload	th:onvolumechange	th:onwaiting
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}" />
```

...with this:

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

5.4 Appending and prepending

Thymeleaf also 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:

```
<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

HTML has the concept of *boolean attributes*, attributes that have no value and the prescence of one means that value is "true". In XHTML, these attributes take just 1 value, which is itself.

For example, `checked`:

```
<input type="checkbox" name="option2" checked /> <!-- HTML -->
<input type="checkbox" name="option1" checked="checked" /> <!-- XHTML -->
```

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 Setting the value of any attribute (default attribute processor)

Thymeleaf offers a *default attribute processor* that allows us to set the value of *any* attribute, even if no specific `th:*` processor has been defined for it at the Standard Dialect.

So something like:

```
<span th:whatever="${user.name}">...</span>
```

Will result in:

```
<span whatever="John Apricot">...</span>
```

5.7 Support for HTML5-friendly attribute and element names

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

```
<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? For that, we will need a way to iterate over items in a collection to build out our product page.

6.1 Iteration basics

To display products in our `/WEB-INF/templates/product/list.html` page we will use 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 repeat 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 method that retrieves the list of products from the service layer and adds it to the template context:

```
public void process(
        final HttpServletRequest request, final HttpServletResponse response,
        final ServletContext servletContext, final ITemplateEngine templateEngine)
        throws Exception {

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

    WebContext ctx = new WebContext(request, response, 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 over the list of products:

```html
<!DOCTYPE html>

<html 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, using the current element in 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 is scoped to the `<tr>` element, which means it is available to inner tags like `<td>`.

Iterable values

The `java.util.List` class isn't the onlyvalue that can be used for iteration in Thymeleaf. 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.Enumeration`.
- Any object implementing `java.util.Iterator`, whose values will be used as they are returned by the iterator, without the need to cache all values in memory.
- 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 with 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>
```

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. Just like the iter variable, the status variable is also scoped to 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:

```html
<!DOCTYPE html>

<html>

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

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>
```

6.3 Optimizing through lazy retrieval of data

Sometimes we might want to optimize the retrieval of collections of data (e.g. from a database) so that these collections are only retrieved if they are really going to be used.

> Actually, this is something that can be applied to *any* piece of data, but given the size that in-memory collections might have, retrieving collections that are meant to be iterated is the most common case for this scenario.

In order to support this, Thymeleaf offers a mechanism to *lazily load context variables*. Context variables that implement the `ILazyContextVariable` interface – most probably by extending its `LazyContextVariable` default implementation – will be resolved in the moment of being executed. For example:

```
context.setVariable(
    "users",
    new LazyContextVariable<List<User>>() {
        @Override
        protected List<User> loadValue() {
            return databaseRepository.findAllUsers();
        }
    });
```

This variable can be used without knowledge of its *lazyness*, in code such as:

```
<ul>
  <li th:each="u : ${users}" th:text="${u.name}">user name</li>
</ul>
```

But at the same time, will never be initialized (its `loadValue()` method will never be called) if `condition` evaluates to `false` in code such as:

```
<ul th:if="${condition}">
  <li th:each="u : ${users}" th:text="${u.name}">user name</li>
</ul>
```

7 Conditional Evaluation

7.1 Simple conditionals: "if" and "unless"

Sometimes you will need a fragment of your template to only 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>
```

This will create 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:

```
<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 an inverse attribute, `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.

```
<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

In our templates, we will often want to include parts from other templates, parts like footers, headers, menus...

In order to do this, Thymeleaf needs us to define these parts, "fragments", for inclusion, which can be done using the `th:fragment` attribute.

Say we want to add a standard copyright footer to all our grocery pages, so we create a `/WEB-INF/templates/footer.html` file containing this code:

```
<!DOCTYPE html>

<html 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:insert` or `th:replace` attributes (and also `th:include`, though its use is no longer recommended since Thymeleaf 3.0):

```
<body>

  ...

  <div th:insert="~{footer :: copy}"></div>

</body>
```

Note that `th:insert` expects a *fragment expression* (`~{...}`), which is *an expression that results in a fragment*. In the above example though, which is a non-complex *fragment expression*, the (`~{ , }`) enclosing is completely optional, so the code above would be equivalent to:

```
<body>

  ...

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

</body>
```

Fragment specification syntax

The syntax of *fragment expressions* is quite straightforward. There are three different formats:

- "`~{templatename::selector}`" Includes the fragment resulting from applying the specified Markup Selector on the template named `templatename` . Note that `selector` can be a mere fragment name, so you could specify something as simple as `~{templatename::fragmentname}` like in the `~{footer :: copy}` above.

  Markup Selector syntax is defined by the underlying AttoParser parsing library, and is similar to XPath expressions or CSS selectors. See [Appendix C](#) for more info.

- "`~{templatename}`" Includes the complete template named `templatename` .

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

- `~{::selector}`" or "`~{this::selector}`" Includes a fragment from the same template.

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

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

Note again how the surrounding `~{...}` envelope is optional in `th:insert` / `th:replace` .

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:insert` / `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 in pages that are perfectly displayable by a browser, with a complete and even *valid* markup structure, while still retaining the ability to make Thymeleaf include them into other templates.

Referencing fragments without `th:fragment`

Thanks to the power of Markup 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:insert="~{footer :: #copy-section}"></div>

</body>
```

Difference between `th:insert` and `th:replace` (and `th:include`)

And what is the difference between `th:insert` and `th:replace` (and `th:include`, not recommended since 3.0)?

- `th:insert` is the simplest: it will simply insert the specified fragment as the body of its host tag.

- `th:replace` actually *replaces* its host tag with the specified fragment.

- `th:include` is similar to `th:insert`, but instead of inserting the fragment it only inserts the *contents* of this fragment.

So an HTML fragment like this:

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

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

```
<body>

  ...

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

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

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

</body>
```

...will result in:

```
<body>

  ...

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

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

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

</body>
```

8.2 Parameterizable fragment signatures

In order to create a more *function-like* mechanism for 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:insert` or `th:replace`:

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

Note that order is not important in the last option:

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

Fragment local variables without fragment arguments

Even if fragments are defined without arguments 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:replace="::frag (onevar=${value1},twovar=${value2})">
```

This would be equivalent to a combination of `th:replace` and `th:with`:

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

Note that this specification of local variables for a fragment – no matter whether it has an argument signature or not – does not cause the context to be emptied prior 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 Flexible layouts: beyond mere fragment insertion

Thanks to *fragment expressions*, we can specify parameters for fragments that are not texts, numbers, bean objects... but instead fragments of markup.

This allows us to create our fragments in a way such that they can be *enriched* with markup coming from the calling templates, resulting in a very flexible **template layout mechanism**.

Note the use of the `title` and `links` variables in the fragment below:

```html
<head th:fragment="common_header(title,links)">

  <title th:replace="${title}">The awesome application</title>

  <!-- Common styles and scripts -->
  <link rel="stylesheet" type="text/css" media="all" th:href="@{/css/awesomeapp.css}">
  <link rel="shortcut icon" th:href="@{/images/favicon.ico}">
  <script type="text/javascript" th:src="@{/sh/scripts/codebase.js}"></script>

  <!--/* Per-page placeholder for additional links */-->
  <th:block th:replace="${links}" />

</head>
```

We can now call this fragment like:

```html
...
<head th:replace="base :: common_header(~{::title},~{::link})">

  <title>Awesome - Main</title>

  <link rel="stylesheet" th:href="@{/css/bootstrap.min.css}">
  <link rel="stylesheet" th:href="@{/themes/smoothness/jquery-ui.css}">

</head>
...
```

...and the result will use the actual `<title>` and `<link>` tags from our calling template as the values of the `title` and `links` variables, resulting in our fragment being customized during insertion:

```html
...
<head>

  <title>Awesome - Main</title>

  <!-- Common styles and scripts -->
  <link rel="stylesheet" type="text/css" media="all" href="/awe/css/awesomeapp.css">
  <link rel="shortcut icon" href="/awe/images/favicon.ico">
  <script type="text/javascript" src="/awe/sh/scripts/codebase.js"></script>

  <link rel="stylesheet" href="/awe/css/bootstrap.min.css">
  <link rel="stylesheet" href="/awe/themes/smoothness/jquery-ui.css">

</head>
...
```

Using the empty fragment

A special fragment expression, the *empty fragment* (`~{}`), can be used for specifying *no markup*. Using the previous example:

```
<head th:replace="base :: common_header(~{::title},~{})">

  <title>Awesome - Main</title>

</head>
...
```

Note how the second parameter of the fragment (`links`) is set to the *empty fragment* and therefore nothing is written for the `<th:block th:replace="${links}" />` block:

```
...
<head>

  <title>Awesome - Main</title>

  <!-- Common styles and scripts -->
  <link rel="stylesheet" type="text/css" media="all" href="/awe/css/awesomeapp.css">
  <link rel="shortcut icon" href="/awe/images/favicon.ico">
  <script type="text/javascript" src="/awe/sh/scripts/codebase.js"></script>

</head>
...
```

Using the no-operation token

The no-op can be also used as a parameter to a fragment if we just want to let our fragment use its current markup as a default value. Again, using the `common_header` example:

```
...
<head th:replace="base :: common_header(_,~{::link})">

  <title>Awesome - Main</title>

  <link rel="stylesheet" th:href="@{/css/bootstrap.min.css}">
  <link rel="stylesheet" th:href="@{/themes/smoothness/jquery-ui.css}">

</head>
...
```

See how the `title` argument (first argument of the `common_header` fragment) is set to *no-op* (`_`), which results in this part of the fragment not being executed at all (`title` = *no-operation*):

```
  <title th:replace="${title}">The awesome application</title>
```

So the result is:

```
...
<head>

  <title>The awesome application</title>

  <!-- Common styles and scripts -->
  <link rel="stylesheet" type="text/css" media="all" href="/awe/css/awesomeapp.css">
  <link rel="shortcut icon" href="/awe/images/favicon.ico">
  <script type="text/javascript" src="/awe/sh/scripts/codebase.js"></script>

  <link rel="stylesheet" href="/awe/css/bootstrap.min.css">
  <link rel="stylesheet" href="/awe/themes/smoothness/jquery-ui.css">

</head>
...
```

Advanced conditional insertion of fragments

The availability of both the *emtpy fragment* and *no-operation token* allows us to perform conditional insertion of fragments in a very easy and elegant way.

For example, we could do this in order to insert our `common :: adminhead` fragment *only* if the user is an administrator, and insert nothing (emtpy fragment) if not:

```
...
<div th:insert="${user.isAdmin()} ? ~{common :: adminhead} : ~{}">...</div>
...
```

Also, we can use the *no-operation token* in order to insert a fragment only if the specified condition is met, but leave the markup without modifications if the condition is not met:

```
...
<div th:insert="${user.isAdmin()} ? ~{common :: adminhead} : _">
    Welcome [[${user.name}]], click <a th:href="@{/support}">here</a> for help-desk support.
</div>
...
```

Additionally, if we have configured our template resolvers to *check for existence* of the template resources — by means of their `checkExistence` flag — we can use the existence of the fragment itself as the condition in a *default* operation:

```
...
<!-- The body of the <div> will be used if the "common :: salutation" fragment  -->
<!-- does not exist (or is empty).                                              -->
<div th:insert="~{common :: salutation} ?: _">
    Welcome [[${user.name}]], click <a th:href="@{/support}">here</a> for help-desk support.
</div>
...
```

8.4 Removing template fragments

Back to the example application, 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:

```html
<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 does that `all` value in the attribute, mean? `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:

```html
<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:

```html
<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 the following works the same as the example above:

```html
<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* the 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 bounds 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 any `<td>` elements.

Thymeleaf offers you a way to declare local variables without iteration, using 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 available for evaluation along with any other variables declared in the context, 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:

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

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 HTML/XML standards do not give any kind of meaning to the order in which the attributes in a tag are written, a *precedence* mechanism had 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:insert
		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):

```html
<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 Thymeleaf, and will be 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 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 with parser-level comment blocks, 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, but not its contents, disappear.

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!

still works OK when open statically in browsers as prototypes!

12 Inlining

12.1 Expression inlining

Although the Standard Dialect allows us to do almost everything 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 `[[...]]` or `[(...)]` are considered **inlined expressions** in Thymeleaf, and inside them we can use any kind of expression that would also be valid in a `th:text` or `th:utext` attribute.

Note that, while `[[...]]` corresponds to `th:text` (i.e. result will be *HTML-escaped*), `[(...)]` corresponds to `th:utext` and will not perform any HTML-escaping. So with a variable such as `msg = 'This is <b>great!</b>'` , given this fragment:

```
<p>The message is "[(${msg})]"</p>
```

The result will have those `<b>` tags unescaped, so:

```
<p>The message is "This is <b>great!</b>"</p>
```

Whereas if escaped like:

```
<p>The message is "[[${msg}]]"</p>
```

The result will be HTML-escaped:

```
<p>The message is "This is &lt;b&gt;great!&lt;/b&gt;"</p>
```

Note that **text inlining is active by default** in the body of every tag in our markup — not the tags themselves —, so there is nothing we need to do to to enable it.

Inlining vs natural templates

If you come from other template engines in which this way of outputting text is the norm, you might 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 design 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 in terms of design usefulness.

Disabling inlining

This mechanism can be disabled though, because there might actually be occasions in which we do want to output the `[[...]]` or `[(...)]` sequences without its contents being processed as an expression. For that, we will use `th:inline="none"`:

```
<p th:inline="none">A double array looks like this: [[1, 2, 3], [4, 5]]!</p>
```

This will result in:

```
<p>A double array looks like this: [[1, 2, 3], [4, 5]]!</p>
```

12.2 Text inlining

Text inlining is very similar to the *expression inlining* capability we have just seen, but it actually adds more power. It has to be enabled explicitly with `th:inline="text"`.

Text inlining not only allows us to use the same *inlined expressions* we just saw, but in fact processes *tag bodies* as if they were templates processed in the `TEXT` template mode, which allows us to perform text-based template logic (not only output expressions).

We will see more about this in the next chapter about the *textual template modes*.

12.3 JavaScript inlining

JavaScript inlining allows for a better integration of JavaScript `<script>` blocks in templates being processed in the `HTML` template mode.

As with *text inlining*, this is actually equivalent to processing the scripts contents as if they were templates in the `JAVASCRIPT` template mode, and therefore all the power of the *textual template modes* (see next chapter) will be at hand. However, in this section we will focus on how we can use it for adding the output of our Thymeleaf expressions into our JavaScript blocks.

This mode has to be explicitly enabled using `th:inline="javascript"`:

```
<script th:inline="javascript">
    ...
    var username = [[${session.user.name}]];
    ...
</script>
```

This will result in:

```
<script th:inline="javascript">
    ...
    var username = "Sebastian \"Fruity\" Applejuice";
    ...
</script>
```

Two important things to note in the code above:

First, that JavaScript inlining will not only output the required text, but also enclose it with quotes and JavaScript-escape its contents, so that the expression results are output as a **well-formed JavaScript literal**.

Second, that this is happening because we are outputting the `${session.user.name}` expression as **escaped**, i.e. using a double-bracket expression: `[[${session.user.name}]]` . If instead we used *unescaped* like:

```
<script th:inline="javascript">
    ...
    var username = [(${session.user.name})];
    ...
</script>
```

The result would look like:

```
<script th:inline="javascript">
    ...
    var username = Sebastian "Fruity" Applejuice;
    ...
</script>
```

…which is malformed JavaScript code. But outputting something unescaped might be what we need if we are building parts of our script by means of appending inlined expressions, so it's good to have this tool at hand.

JavaScript natural templates

The mentioned *intelligence* of the JavaScript inlining mechanism goes much further than just applying JavaScript-specific escaping and outputting expression results as valid literals.

For example, we can wrap our (escaped) inlined expressions in JavaScript comments like:

```
<script th:inline="javascript">
    ...
    var username = /*[[${session.user.name}]]*/ "Gertrud Kiwifruit";
    ...
</script>
```

And Thymeleaf will ignore everything we have written *after the comment and before the semicolon* (in this case `'Gertrud Kiwifruit'`), so the result of executing this will look exactly like when we were not using the wrapping comments:

```
<script th:inline="javascript">
    ...
    var username = "Sebastian \"Fruity\" Applejuice";
    ...
</script>
```

But have another careful look at the original template code:

```
<script th:inline="javascript">
    ...
    var username = /*[[${session.user.name}]]*/ "Gertrud Kiwifruit";
    ...
</script>
```

Note how this is **valid JavaScript** code. And it will perfectly execute when you open your template file in a static manner (without executing it at a server).

So what we have here is a way to do **JavaScript natural templates**!

Advanced inlined evaluation and JavaScript serialization

An important thing to note regarding JavaScript inlining is that this expression evaluation is intelligent and not limited to Strings. Thymeleaf will correctly write in JavaScript 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">
    ...
    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">
    ...
    var user = {"age":null,"firstName":"John","lastName":"Apricot",
                "name":"John Apricot","nationality":"Antarctica"};
    ...
</script>
```

The way this JavaScript serialization is done is by means of an implementation of the `org.thymeleaf.standard.serializer.IStandardJavaScriptSerializer` interface, which can be configured at the instance of the `StandardDialect` being used at the template engine.

The default implementation of this JS serialization mechanism will look for the Jackson library in the classpath and, if present, will use it. If not, it will apply a built-in serialization mechanism that covers the needs of most scenarios and produces similar results (but is less flexible).

12.4 CSS inlining

Thymeleaf also allows the use of inlining in CSS `<style>` tags, such as:

```
<style th:inline="css">
   ...
</style>
```

For example, say we have two variables set to two different `String` values:

```
classname = 'main elems'
align = 'center'
```

We could use them just like:

```
<style th:inline="css">
    .[[${classname}]] {
      text-align: [[${align}]];
    }
</style>
```

And the result would be:

```
<style th:inline="css">
    .main\ elems {
      text-align: center;
    }
</style>
```

Note how CSS inlining also bears some *intelligence*, just like JavaScript's. Specifically, expressions output via *escaped* expressions like `[[${classname}]]` will be escaped as **CSS identifiers**. That is why our `classname = 'main elems'` has turned into `main\ elems` in the fragment of code above.

Advanced features: CSS natural templates, etc.

In an equivalent way to what was explained before for JavaScript, CSS inlining also allows for our `<style>` tags to work both statically and dynamically, i.e. as **CSS natural templates** by means of wrapping inlined expressions in comments. See:

```
<style th:inline="css">
    .main\ elems {
      text-align: /*[[${align}]]*/ left;
    }
</style>
```

13 Textual template modes

13.1 Textual syntax

Three of the Thymeleaf *template modes* are considered **textual**: TEXT , JAVASCRIPT and CSS . This differentiates them from the markup template modes: HTML and XML .

The key difference between *textual* template modes and the markup ones is that in a textual template there are no tags into which to insert logic in the form of attributes, so we have to rely on other mechanisms.

The first and most basic of these mechanisms is **inlining**, which we have already detailed in the previous chapter. Inlining syntax is the most simple way to output results of expressions in textual template mode, so this is a perfectly valid template for a text email.

```
Dear [(${name})],

Please find attached the results of the report you requested
with name "[(${report.name})]".

Sincerely,
  The Reporter.
```

Even without tags, the example above is a complete and valid Thymeleaf template that can be executed in the TEXT template mode.

But in order to include more complex logic than mere *output expressions*, we need a new non-tag-based syntax:

```
[# th:each="item : ${items}"]
  - [(${item})]
[/]
```

Which is actually the *condensed* version of the more verbose:

```
[#th:block th:each="item : ${items}"]
  - [#th:block th:utext="${item}" /]
[/th:block]
```

Note how this new syntax is based on elements (i.e. processable tags) that are declared as [#element ...] instead of <element ...> . Elements are open like [#element ...] and closed like [/element] , and standalone tags can be declared by minimizing the open element with a / in a way almost equivalent to XML tags: [#element ... /] .

The Standard Dialect only contains a processor for one of these elements: the already-known th:block , though we could extend this in our dialects and create new elements in the usual way. Also, the th:block element ([#th:block ...] ... [/th:block]) is allowed to be abbreviated as the empty string ([# ...] ... [/]), so the above block is actually equivalent to:

```
[# th:each="item : ${items}"]
  - [# th:utext="${item}" /]
[/]
```

And given [# th:utext="${item}" /] is equivalent to an *inlined unescaped expression*, we could just use it in order to have less code. Thus we end up with the first fragment of code we saw above:

```
[# th:each="item : ${items}"]
  - [(${item})]
[/]
```

Note that the *textual syntax requires full element balance (no unclosed tags) and quoted attributes* – it's more XML-style than HTML-style.

Let's have a look at a more complete example of a TEXT template, a *plain text* email template:

```
Dear [(${customer.name})],

This is the list of our products:

[# th:each="prod : ${products}"]
   - [(${prod.name})]. Price: [(${prod.price})] EUR/kg
[/]

Thanks,
  The Thymeleaf Shop
```

After executing, the result of this could be something like:

```
Dear Mary Ann Blueberry,

This is the list of our products:

   - Apricots. Price: 1.12 EUR/kg
   - Bananas. Price: 1.78 EUR/kg
   - Apples. Price: 0.85 EUR/kg
   - Watermelon. Price: 1.91 EUR/kg

Thanks,
  The Thymeleaf Shop
```

And another example in JAVASCRIPT template mode, a greeter.js file, we process as a textual template and which result we call from our HTML pages. Note this is *not* a <script> block in an HTML template, but a .js file being processed as a template on its own:

```
var greeter = function() {

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

    [# th:each="salut : ${salutations}"]
      alert([[${salut}]] + " " + username);
    [/]

};
```

After executing, the result of this could be something like:

```
var greeter = function() {

    var username = "Bertrand \"Crunchy\" Pear";

    alert("Hello" + " " + username);
    alert("Ol\u00E1" + " " + username);
    alert("Hola" + " " + username);

};
```

Escaped element attributes

In order to avoid interactions with parts of the template that might be processed in other modes (e.g. `text` -mode inlining inside an `HTML` template), Thymeleaf 3.0 allows the attributes in elements in its *textual syntax* to be escaped. So:

- Attributes in `TEXT` template mode will be *HTML-unescaped*.
- Attributes in `JAVASCRIPT` template mode will be *JavaScript-unescaped*.
- Attributes in `CSS` template mode will be *CSS-unescaped*.

So this would be perfectly OK in a `TEXT` -mode template (note the `>`):

```
[# th:if="${120&lt;user.age}"]
   Congratulations!
[/]
```

Of course that `<` would make no sense in a *real text* template, but it is a good idea if we are processing an HTML template with a `th:inline="text"` block containing the code above and we want to make sure our browser doesn't take that `<user.age` for the name of an open tag when statically opening the file as a prototype.

13.2 Extensibility

One of the advantages of this syntax is that it is just as extensible as the *markup* one. Developers can still define their own dialects with custom elements and attributes, apply a prefix to them (optionally), and then use them in textual template modes:

```
[#myorg:dosomething myorg:importantattr="211"]some text[/myorg:dosomething]
```

13.3 Textual prototype-only comment blocks: adding code

The `JAVASCRIPT` and `CSS` template modes (not available for `TEXT`) allow including code between a special comment syntax `/*[+...+]*/` so that Thymeleaf will automatically uncomment such 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() {
...
```

13.4 Textual parser-level comment blocks: removing code

In a way similar to that of prototype-only comment blocks, all the three textual template modes (TEXT , JAVASCRIPT and CSS) make it possible to instruct Thymeleaf to remove code between special /*[- */ and /* -]*/ marks, like this:

```
var x = 23;

/*[- */

var msg  = "This is shown only when executed statically!";

/* -]*/

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

Or this, in TEXT mode:

```
...
/*[- Note the user is obtained from the session, which must exist -]*/
Welcome [(${session.user.name})]!
...
```

13.5 Natural JavaScript and CSS templates

As seen in the previous chapter, JavaScript and CSS inlining offer the possibility to include inlined expressions inside JavaScript/CSS comments, like:

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

...which is valid JavaScript, and once executed could look like:

```
...
var username = "John Apricot";
...
```

This same *trick* of enclosing inlined expressions inside comments can in fact be used for the entire textual mode syntax:

```
/*[# th:if="${user.admin}"]*/
    alert('Welcome admin');
/*[/]*/
```

That alert in the code above will be shown when the template is open statically – because it is 100% valid JavaScript –, and also when the template is run if the user is an admin. It is equivalent to:

```
[# th:if="${user.admin}"]
    alert('Welcome admin');
[/]
```

...which is actually the code to which the initial version is converted during template parsing.

Note however that wrapping elements in comments does not clean the lines they live in (to the right until a ; is found) as inlined output expressions do. That behaviour is reserved for inlined output expressions only.

So Thymeleaf 3.0 allows the development of **complex JavaScript scripts and CSS style sheets in the form of natural templates**, valid both as a *prototype* and as a *working template*.

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 HTML 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` :

```html
<!DOCTYPE html>

<html 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:

```html
<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:

```html
<!DOCTYPE html>

<html 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}` 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 us the ability to create our 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(template);
```

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

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

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

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

- `org.thymeleaf.templateresolver.StringTemplateResolver` , which resolves templates directly as the `String` being specified as `template` (or *template name*, which in this case is obviously much more than a mere name):

```
return new StringReader(templateName);
```

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);
```

The Thymeleaf + Spring integration packages offer a `SpringResourceTemplateResolver` implementation which uses all the Spring infrastructure for accessing and reading resources in applications, and which is the recommended implementation in Spring-enabled applications.

Chaining Template Resolvers

Also, a Template Engine can specify 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(servletContext);
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 a recommendation:

```
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(servletContext);
servletContextTemplateResolver.setOrder(Integer.valueOf(2));
```

If these *resolvable patterns* are not specified, we will be relying on the specific capabilities of each of the `ITemplateResolver` implementations we are using. Note that not all implementations might be able to determine the existence of a template before resolving, and thus could always consider a template as *resolvable* and break the resolution chain (not allowing other resolvers to check for the same template), but then be unable to read the real resource.

All the `ITemplateResolver` implementations that are included with core Thymeleaf include a mechanism that will allow us to make the resolvers *really check* if a resource exists before considering it *resolvable*. It is the `checkExistence` flag, which works like:

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

This `checkExistence` flag forces the resolver perform a *real check* for resource existence during the resolution phase (and let the following resolver in the chain be called if existence check returns false). While this might sound good in every case, in most cases this will mean a double access to the resource itself (once for checking existence, another time for reading it), and could be a performance issue in some scenarios, e.g. remote URL-based template resources – a potential performance issue that might anyway get largely mitigated by the use of the template cache (in which case templates will only be *resolved* the first time they are accessed).

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.

`StandardMessageResolver` is the standard implementation of the `IMessageResolver` interface, but we could create our own if we wanted, adapted to the specific needs of our application.

> The Thymeleaf + Spring integration packages offer by default an `IMessageResolver` implementation which uses the standard Spring way of retrieving externalized messages, by using `MessageSource` beans declared at the Spring Application Context.

Standard Message Resolver

So how does `StandardMessageResolver` look for the messages requested at a specific template?

If the template name is `home` and it is located in `/WEB-INF/templates/home.html`, and the requested locale is `gl_ES` then this resolver will look for messages in the following files, in this order:

- `/WEB-INF/templates/home_gl_ES.properties`
- `/WEB-INF/templates/home_gl.properties`
- `/WEB-INF/templates/home.properties`

Refer to the JavaDoc documentation of the `StandardMessageResolver` class for more detail on how the complete message resolution mechanism works.

Configuring message resolvers

What if we 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 we 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 Conversion Services

The *conversion service* that enables us to perform data conversion and formatting operations by means of the *double-brace* syntax (${{...}}) is actually a feature of the Standard Dialect, not of the Thymeleaf Template Engine itself.

As such, the way to configure it is by setting our custom implementation of the `IStandardConversionService` interface directly into the instance of `StandardDialect` that is being configured into the template engine. Like:

```
IStandardConversionService customConversionService = ...

StandardDialect dialect = new StandardDialect();
dialect.setConversionService(customConversionService);

templateEngine.setDialect(dialect);
```

Note that the thymeleaf-spring3 and thymeleaf-spring4 packages contain the `SpringStandardDialect`, and this dialect already comes pre-configured with an implementation of `IStandardConversionService` that integrates Spring's own *Conversion Service* infrastructure into Thymeleaf.

15.4 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 we might want to use in our application (for example, `log4j`).

Thymeleaf classes will log `TRACE` , `DEBUG` and `INFO` -level information, depending on the level of detail we desire, and besides general logging it will use three special loggers associated with the TemplateEngine class which we 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.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 set of parsers – for markup and text – that parse templates into sequences of events (open tag, text, close tag, comment, etc.) and a series of processors – one for each type of behaviour that needs to be applied – that modify the template parsed event sequence in order to create the results we expect by combining the original template with our data.

It also includes – by default – a cache that stores parsed templates; the sequence of events 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 processing is extremely quick by comparison.
* Cloning an existing in-memory event sequence is always much quicker than reading a template file, parsing it and creating a new event sequence for it.
* Web applications usually have only 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 large 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 Decoupled Template Logic

17.1 Decoupled logic: The concept

So far we have worked for our Grocery Store with templates done the *usual way*, with logic being inserted into our templates in the form of attributes.

But Thymeleaf also allows us to completely *decouple* the template markup from its logic, allowing the creation of **completely logic-less markup templates** in the `HTML` and `XML` template modes.

The main idea is that template logic will be defined in a separate *logic file* (more exactly a *logic resource*, as it doesn't need to be a *file*). By default, that logic resource will be an additional file living in the same place (e.g. folder) as the template file, with the same name but with `.th.xml` extension:

```
/templates
+->/home.html
+->/home.th.xml
```

So the `home.html` file can be completely logic-less. It might look like this:

```html
<!DOCTYPE html>
<html>
  <body>
    <table id="usersTable">
      <tr>
        <td class="username">Jeremy Grapefruit</td>
        <td class="usertype">Normal User</td>
      </tr>
      <tr>
        <td class="username">Alice Watermelon</td>
        <td class="usertype">Administrator</td>
      </tr>
    </table>
  </body>
</html>
```

Absolutely no Thymeleaf code there. This is a template file that a designer with no Thymeleaf or templating knowledge could have created, edited and/or understood. Or a fragment of HTML provided by some external system with no Thymeleaf hooks at all.

Let's now turn that `home.html` template into a Thymeleaf template by creating our additional `home.th.xml` file like this:

```xml
<?xml version="1.0"?>
<thlogic>
  <attr sel="#usersTable" th:remove="all-but-first">
    <attr sel="/tr[0]" th:each="user : ${users}">
      <attr sel="td.username" th:text="${user.name}" />
      <attr sel="td.usertype" th:text="#{|user.type.${user.type}|}" />
    </attr>
  </attr>
</thlogic>
```

Here we can see a lot of `<attr>` tags inside a `thlogic` block. Those `<attr>` tags perform *attribute injection* on nodes of the original template selected by means of their `sel` attributes, which contain Thymeleaf *markup selectors* (actually *AttoParser markup selectors*).

Also note that `<attr>` tags can be nested so that their selectors are *appended*. That `sel="/tr[0]"` above, for example, will be processed as `sel="#usersTable/tr[0]"`. And the selector for the user name `<td>` will be processed as `sel="#usersTable/tr[0]//td.username"`.

So once merged, both files seen above will be the same as:

```html
<!DOCTYPE html>
<html>
  <body>
    <table id="usersTable" th:remove="all-but-first">
      <tr th:each="user : ${users}">
        <td class="username" th:text="${user.name}">Jeremy Grapefruit</td>
        <td class="usertype" th:text="#{|user.type.${user.type}|}">Normal User</td>
      </tr>
      <tr>
        <td class="username">Alice Watermelon</td>
        <td class="usertype">Administrator</td>
      </tr>
    </table>
  </body>
</html>
```

This looks more familiar, and is indeed less *verbose* than creating two separate files. But the advantage of *decoupled templates* is that we can give for our templates total independence from Thymeleaf, and therefore better maintainability from the design standpoint.

Of course some *contracts* between designers or developers will still be needed – e.g. the fact that the users `<table>` will need an `id="usersTable"` –, but in many scenarios a pure-HTML template will be a much better communication artifact between design and development teams.

17.2 Configuring decoupled templates

Enabling decoupled templates

Decoupled logic will not be expected for every template by default. Instead, the configured template resolvers (implementations of `ITemplateResolver`) will need to specifically mark the templates they resolve as *using decoupled logic*.

Except for `StringTemplateResolver` (which does not allow decoupled logic), all other out-of-the-box implementations of `ITemplateResolver` will provide a flag called `useDecoupledLogic` that will mark all templates resolved by that resolver as potentially having all or part of its logic living in a separate resource:

```java
final ServletContextTemplateResolver templateResolver =
        new ServletContextTemplateResolver(servletContext);
...
templateResolver.setUseDecoupledLogic(true);
```

Mixing coupled and decoupled logic

Decoupled template logic, when enabled, is not a requirement. When enabled, it means that the engine will *look for* a resource containing decoupled logic, parsing and merging it with the original template if it exists. No error will be thrown if the decoupled logic resource does not exist.

Also, in the same template we can mix both *coupled* and *decoupled* logic, for example by adding some Thymeleaf attributes at the original template file but leaving others for the separate decoupled logic file. The most common case for

this is using the new (in v3.0) `th:ref` attribute.

17.3 The th:ref attribute

`th:ref` is only a marker attribute. It does nothing from the processing standpoint and simply disappears when the template is processed, but its usefulness lies in the fact that it acts as a *markup reference*, i.e. it can be resolved by name from a *markup selector* just like a *tag name* or a *fragment* (`th:fragment`).

So if we have a selector like:

```
<attr sel="whatever" .../>
```

This will match:

* Any `<whatever>` tags.
* Any tags with a `th:fragment="whatever"` attribute.
* Any tags with a `th:ref="whatever"` attribute.

What is the advantage of `th:ref` against, for example, using a pure-HTML `id` attribute? Merely the fact that we might not want to add so many `id` and `class` attributes to our tags to act as *logic anchors*, which might end up *polluting* our output.

And in the same sense, what is the disadvantage of `th:ref` ? Well, obviously that we'd be adding a bit of Thymeleaf logic (*"logic"*) to our templates.

Note this applicability of the `th:ref` attribute **does not only apply to decoupled logic template files** : it works the same in other types of scenarios, like in fragment expressions (`~{...}`).

17.4 Performance impact of decoupled templates

The impact is extremely small. When a resolved template is marked to use decoupled logic and it is not cached, the template logic resource will be resolved first, parsed and processed into a secuence of instructions in-memory: basically a list of attributes to be injected to each markup selector.

But this is the only *additional step* required because, after this, the real template will be parsed, and while it is parsed these attributes will be injected *on-the-fly* by the parser itself, thanks to the advanced capabilities for node selection in AttoParser. So parsed nodes will come out of the parser as if they had their injected attributes written in the original template file.

The biggest advantage of this? When a template is configured to be cached, it will be cached already containing the injected attributes. So the overhead of using *decoupled templates* for cacheable templates, once they are cached, will be absolutely *zero*.

17.5 Resolution of decoupled logic

The way Thymeleaf resolves the decoupled logic resources corresponding to each template is configurable by the user. It is determined by an extension point, the `org.thymeleaf.templateparser.markup.decoupled.IDecoupledTemplateLogicResolver` , for which a *default implementation* is provided: `StandardDecoupledTemplateLogicResolver` .

What does this standard implementation do?

- First, it applies a `prefix` and a `suffix` to the *base name* of the template resource (obtained by means of its `ITemplateResource#getBaseName()` method). Both prefix and suffix can be configured and, by default, the prefix will be empty and the suffix will be `.th.xml` .
- Second, it asks the template resource to resolve a *relative resource* with the computed name by means of its `ITemplateResource#relative(String relativeLocation)` method.

The specific implementation of `IDecoupledTemplateLogicResolver` to be used can be configured at the `TemplateEngine` easily:

```
final StandardDecoupledTemplateLogicResolver decoupledresolver =
        new StandardDecoupledTemplateLogicResolver();
decoupledResolver.setPrefix("../viewlogic/");
...
templateEngine.setDecoupledTemplateLogicResolver(decoupledResolver);
```

18 Appendix A: Expression Basic Objects

Some objects and variable maps are always available to be invoked. Let's see them:

Base objects

- **#ctx** : the context object. An implementation of `org.thymeleaf.context.IContext` or `org.thymeleaf.context.IWebContext` depending on our environment (standalone or web).

  Note `#vars` and `#root` are synomyns for the same object, but using `#ctx` is recommended.

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

${#ctx.locale}
${#ctx.variableNames}

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

${#ctx.request}
${#ctx.response}
${#ctx.session}
${#ctx.servletContext}
```

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

```
${#locale}
```

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, 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):

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

```
${#request.getAttribute('foo')}
${#request.getParameter('foo')}
${#request.getContextPath()}
${#request.getRequestName()}
...
```

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

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

- **#servletContext** : direct access to the `javax.servlet.ServletContext` object associated with the current request.

```
${#servletContext.getAttribute('foo')}
${#servletContext.contextPath}
...
```

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

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

19 Appendix B: Expression Utility Objects

Execution Info

- **#execInfo** : expression object providing useful information about the template being processed inside Thymeleaf Standard Expressions.

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

/*
 * Return the name and mode of the 'leaf' template. This means the template
 * from where the events being processed were parsed. So if this piece of
 * code is not in the root template "A" but on a fragment being inserted
 * into "A" from another template called "B", this will return "B" as a
 * name, and B's mode as template mode.
 */
${#execInfo.templateName}
${#execInfo.templateMode}

/*
 * Return the name and mode of the 'root' template. This means the template
 * that the template engine was originally asked to process. So if this
 * piece of code is not in the root template "A" but on a fragment being
 * inserted into "A" from another template called "B", this will still
 * return "A" and A's template mode.
 */
${#execInfo.processedTemplateName}
${#execInfo.processedTemplateMode}

/*
 * Return the stacks (actually, List<String> or List<TemplateMode>) of
 * templates being processed. The first element will be the
 * 'processedTemplate' (the root one), the last one will be the 'leaf'
 * template, and in the middle all the fragments inserted in nested
 * manner to reach the leaf from the root will appear.
 */
${#execInfo.templateNames}
${#execInfo.templateModes}

/*
 * Return the stack of templates being processed similarly (and in the
 * same order) to 'templateNames' and 'templateModes', but returning
 * a List<TemplateData> with the full template metadata.
 */
${#execInfo.templateStack}
```

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)}
```

URIs/URLs

- **#uris** : utility object for performing URI/URL operations (esp. escaping/unescaping) inside Thymeleaf Standard Expressions.

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

/*
 * Escape/Unescape as a URI/URL path
 */
${#uris.escapePath(uri)}
${#uris.escapePath(uri, encoding)}
${#uris.unescapePath(uri)}
${#uris.unescapePath(uri, encoding)}

/*
 * Escape/Unescape as a URI/URL path segment (between '/' symbols)
 */
${#uris.escapePathSegment(uri)}
${#uris.escapePathSegment(uri, encoding)}
${#uris.unescapePathSegment(uri)}
${#uris.unescapePathSegment(uri, encoding)}

/*
 * Escape/Unescape as a Fragment Identifier (#frag)
 */
${#uris.escapeFragmentId(uri)}
${#uris.escapeFragmentId(uri, encoding)}
${#uris.unescapeFragmentId(uri)}
${#uris.unescapeFragmentId(uri, encoding)}

/*
 * Escape/Unescape as a Query Parameter (?var=value)
 */
${#uris.escapeQueryParam(uri)}
${#uris.escapeQueryParam(uri, encoding)}
${#uris.unescapeQueryParam(uri)}
${#uris.unescapeQueryParam(uri, encoding)}
```

Conversions

- **#conversions** : utility object that allows the execution of the *Conversion Service* at any point of a template:

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

/*
 * Execute the desired conversion of the 'object' value into the
 * specified class.
 */
${#conversions.convert(object, 'java.util.TimeZone')}
${#conversions.convert(object, targetClass)}
```

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 ISO8601 format
 * Also works with arrays, lists or sets
 */
${#dates.formatISO(date)}
${#dates.arrayFormatISO(datesArray)}
${#dates.listFormatISO(datesList)}
${#dates.setFormatISO(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 ISO8601 format
 * Also works with arrays, lists or sets
 */
${#calendars.formatISO(cal)}
${#calendars.arrayFormatISO(calArray)}
${#calendars.listFormatISO(calList)}
${#calendars.setFormatISO(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', 'WHITESPACE', '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(first, second)}
${#strings.equalsIgnoreCase(first, second)}
${#strings.concat(values...)}
${#strings.concatReplaceNulls(nullValue, values...)}


/*
 * 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)}
```

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: Markup Selector Syntax

Thymeleaf's Markup Selectors are directly borrowed from Thymeleaf's parsing library: AttoParser.

The syntax for this selectors has large similarities with that of selectors in XPath, CSS and jQuery, which makes them easy to use for most users. You can have a look at the complete syntax reference at the AttoParser documentation.

For example, the following selector will select every `<div>` with the class `content`, in every position inside the markup (note this is not as concise as it could be, read on to know why):

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

The basic syntax 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, a *reference* being a `th:ref` or a `th:fragment` attribute).

- 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 `<x>` tags that have a `th:ref="oneref"` or `th:fragment="oneref"` attribute.

- `%oneref` means any tags that have a `th:ref="oneref"` or `th:fragment="oneref"` attribute. 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']`.

So the above Markup Selector expression:

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

Could be written as:

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

Examining a different example, this:

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

Will look for a `th:fragment="myfrag"` fragment signature (or `th:ref` references). 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 (or `th:ref` references).

Multivalued class matching

Markup 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.two` will match `<div class="one two three" />`