Tom Gardiner Follow
Founder @ Trevor.io. Formerly Co-founder and CTO @ RefME. Background in High Performance Computing and Artificial Intelligence.
Apr 4, 2016 · 11 min read

Thymeleaf with Spring MVC (rapid introduction to the essentials)

Thymeleaf has quickly become the de-facto server-side (HTML) template engine for Spring MVC. It has a wealth of features, extensions, bells and whistles. However, to get started, you only really need to know a few bits and pieces.

I will provide a rapid introduction to those here.

Adding Thymeleaf to your project

> *Note: I assume you are using* Spring Boot *with* Gradle *as your build tool. If you are not, you may need to perform a few additional steps, but the key takeaways here will still apply.*

Adding Thymeleaf is as simple as adding it to your *build.gradle* file:

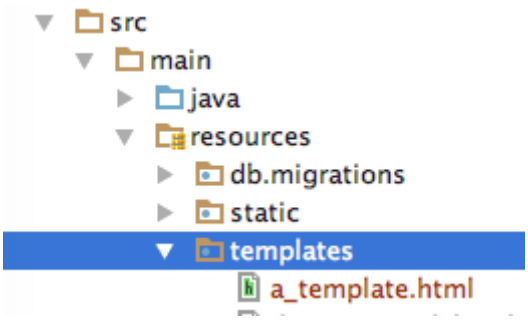
```
compile('org.springframework.boot:spring-boot-starter-
thymeleaf:1.3.1.RELEASE')
```

and for the purpose of demonstrating functionality, we will also add the Twitter Bootstrap webjar:

```
compile('org.webjars:bootstrap:3.3.6')
```

Great! You now have Thymeleaf as your server-side HTML template engine. Let's get started.

Thymeleaf automatically picks up your template files from your *resources/templates* directory:



so make sure to put all your templates in here. Just name them whatever.html. Thymeleaf templates are also valid html files.

You will want to turn off the Thymeleaf cache during development; as otherwise you will regularly be looking at stale versions of the templates.

This can easily be configured in your *application.properties* file:

```
spring.thymeleaf.cache=false
```

Creating a default layout template

We'll start by creating a *default layout template* that will provide a standard structure for all our pages.

This is often good practice, as it means we don't have to redefine the boilerplate layout again and again in each new page we create. We just focus on the content that is unique to that page. This *default layout template* will import all the standard CSS and javascript resources that we always want access to; and if this set changes in the future (e.g. we want to add google analytics code to every page) we only have to add it in one place.

We will name our default layout template *default.html* and add it to our *resources/templates* directory.

Here is its content:

```
<!DOCTYPE html>
<html lang="en"
      xmlns:th="http://www.thymeleaf.org"

xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout">
    <head>
```

```
        <meta charset="UTF-8"/>
        <title>Default title for my pages</title>

        <link rel="stylesheet"
href="/webjars/bootstrap/3.3.6/css/bootstrap.min.css"/>
        <link rel="stylesheet"
href="/webjars/bootstrap/3.3.6/css/bootstrap-
theme.min.css"/>
    </head>
    <body>
```

```
        <div class="container">
            <div class="row">
                <div class="col-md-3"></div>
                <div class="col-md-6"
layout:fragment="content">
                        Page content goes here
                </div>
                <div class="col-md-3"></div>
            </div>
        </div>

        <script src="/webjars/jquery/1.11.1/jquery.min.js">
</script>
        <script
src="/webjars/bootstrap/3.3.6/js/bootstrap.min.js"></script>
        <th:block layout:fragment="scripts"></th:block>
    </body>
</html>
```

As you can see from the content of our *body* tag, we have used bootstrap to define a very basic structure for our website, made up of 3 columns, where only the middle column contains any content. This gives us a basic centred look with space left and right of our content.

A few things you should notice:

- The Thymeleaf namespaces (*th* and *layout*) have been defined at the top of the template. These are important, as we will be using them throughout.
- We have given the page a default title ("*Default title for my pages*"). This isn't actually necessary as long as you override it in all your pages, but having one here can be useful in case many of your pages want to use the default title.
- By including Twitter Bootstrap in our gradle file earlier, the bootstrap resources are automatically made available in the */webjars/* folder. We then include them here so that they are available in all our pages.
- At the bottom of the page we have included our javascript files. The first two are the bootstrap and jquery (both in the webjars

folder). The third is a *fragment* for our own needs. We will use this later to add custom javascript files to our pages.

- Finally, notice that the middle column in our layout is also a *fragment* called "*content*". This is the fragment we will provide the content for in each of our pages, in order to populate that middle column with unique content for that page.

Hello World page

Right, we're ready to create our first page.

Let's call it *hello.html* and again put it in the *resources/templates* directory. It will look as follows:

```
<!DOCTYPE html>
<html lang="en"
      xmlns:th="http://www.thymeleaf.org"

xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      layout:decorator="default">
    <head>
        <title>Hello world page</title>
        <link rel="stylesheet" href="/css/hello.css"/>
    </head>
    <body>
        <div layout:fragment="content">
            <h1>Hello world</h1>
        </div>
    </body>
    <th:block layout:fragment="scripts">
        <script src="/js/hello.js"></script>
    </th:block>
</html>
```

The first thing you will probably observe is that even though we have already added the html, head and body tags in our default.html template, we add them again here. This is a feature of Thymeleaf: it is designed so that every individual template is actually perfectly renderable by itself. This is so that any client-side developer/designer can take any page and just edit/open them in their browser without needing a running server. Very useful for teams.

The key things to notice from this example:

- Near the top, we have have specified that our "*decorator*" (default layout template) is the *default.html* template. This will tell the Thymeleaf rendering engine to use our default.html template to decorate this page (when run on the server).

- We have overridden the page title by adding it to the *HEAD* tag.
- We have added an **additional** css file to our page (*/css/hello.css*). Note: this will **not** replace the css files included in default.html, but just add to them.
- We have defined the "*content*" fragment (declared in our default.html template) by adding a H1 Hello World element. This is the content that will be displayed in the central column of our page's content.
- And at the bottom of the page we have added an **additional** js file. We could also have added this to our HEAD element, but the problem is that if our javascript file wanted access to, for example, *jquery*, then it would fail, as *jquery* isn't defined until the end of the page. By adding it to this fragment, our javascript file (*/js/hello.js*) is the last file to be evaluated by the browser (after jquery and bootstrap are initialised).

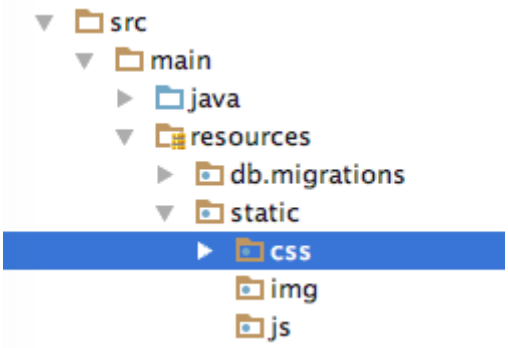
Good stuff; we now have a standard template for all our pages, as well as a hello world example that uses it. But how do we test it?

Well, we still have a few more steps to go. Firstly, the hello.css and hello.js files that we're referencing need to be put somewhere, and secondly, we need a controller to render the page.

Let's do those steps now.

Static assets

The static assets simply go in the *resources/static* directory, next to the *resources/templates* directory:



As the focus of this blog post is HTML templates, rather than CSS/JS, this is the last mention I'll make to any css/js files. Basically: by adding your css and js files to the directories shown above, Spring MVC will

automatically make them available at */css/*.css* and */js/*.js* respectively.

Hello World Controller

The final thing we need is a controller to render the template.

Spring MVC makes it very simple to make controllers:

```
@Controller
public class HelloController {

    @RequestMapping("/hello")
    public String hello() {
        return "hello";
    }
}
```

Only two things you need to notice:

- The controller maps requests for */hello* to this page (so if run locally, the page will be available at http://localhost:8080/hello)
- The controller returns "*hello*" which means "*render the hello.html template*". Thymeleaf knows to find this in the *resources/templates* directory.

And that's it. Hello world working. Start up your Spring MVC server and direct your browser to http://localhost:8080/hello.

You will see that the following page content will be rendered:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Hello world page</title>
        <meta charset="UTF-8" />
        <link rel="stylesheet"
href="/webjars/bootstrap/3.3.6/css/bootstrap.min.css" />
        <link rel="stylesheet"
href="/webjars/bootstrap/3.3.6/css/bootstrap-theme.min.css"
/>
        <link rel="stylesheet" href="/css/hello.css" />
    </head>
    <body>
        <div class="container">
            <div class="row">
                <div class="col-md-3"></div>
                <div class="col-md-6">
```

```
                    <h1>Hello world</h1>
                </div>
                <div class="col-md-3"></div>
            </div>
        </div>
        <script src="/webjars/jquery/1.11.1/jquery.min.js">
</script>
        <script
src="/webjars/bootstrap/3.3.6/js/bootstrap.min.js"></script>
        <script src="/js/hello.js"></script>
    </body>
</html>
```

Notice how the content of default.html and hello.html have been merged together, as expected.

Injecting model parameters

Next let's inject in some content from the server.

Update your controller to be as follows:

```
@RequestMapping("/hello")
public String hello(Model model) {
    model.addAttribute("name", "Tom");
    model.addAttribute("formatted", "<b>blue</b>");
    return "hello";
}
```

and update the "*content*" fragment in your *hello.html* file to be the following:

```
<div layout:fragment="content">
    <span th:text="|Hello ${name}|"/>, did you know that
<span th:utext="${formatted}"/> is <span
th:text="${formatted}"/> in HTML?
</div>
```

Here we have provided two attributes to our model ("name" and "formatted") and have rendered them as part of our content. The end result will look something like this:

Hello Tom, did you know that **blue** is blue in HTML?

We have applied a number of techniques here:

- the first is th:text which tells Thymeleaf: **replace** the content of this element with the value in quotes.
- the second is $ syntax (e.g. th:text="${formatted}"). This allows us to take model attributes and place them directly into our content.
- The third is "|…|" syntax (e.g. th:text="|Hello ${name}|"). This allows us to do String interpolation, whereby model variables are placed *inside* normal text. This contrasts with th:text="${formatted}" where we just use the value of the model attribute (no string interpolation).
- The fourth is th:utext which tells Thymeleaf: do not escape the text being injected (i.e. Un-escaped Text); hence why **blue** is actually made bold above.

That's the basic principle for injecting model parameters into Thymeleaf templates. Next we'll look at something a little more powerful.

Dynamic lists/rows of content

Often you'll have a list or array of items that you want to render on your page. Thymeleaf makes this easy.

First let's update our HelloController to contain a list of items:

```
@Controller
public class HelloController {

    @RequestMapping("/hello")
    public String hello(Model model) {
        DeveloperResource[] devResources = {
                new DeveloperResource("Google",
                     "http://www.google.com"),
                new DeveloperResource("Stackoverflow",
                     "http://www.stackoverflow.com"),
                new DeveloperResource("W3Schools",
                    "http://www.w3schools.com")
        };
        model.addAttribute("resources", devResources);
        return "hello";
    }

    public static final class DeveloperResource {
        private final String name;
        private final String url;
```

```
        public DeveloperResource(String name, String url) {
            this.name = name;
            this.url = url;
        }

        public String getName() {
            return name;
        }

        public String getUrl() {
            return url;
        }
    }
}
```

Now update our hello.html template to render this list:

```
<div layout:fragment="content">
    <div class="list-group">
        <a th:each="resource : ${resources}"
           th:href="${resource.url}" class="list-group-
item">
            <b th:text="${resource.name}"></b>
        </a>
    </div>
</div>
```

Loading this in your browser, the rendered html will look like this:

```
<div class="list-group">
    <a class="list-group-item" href="http://www.google.com">
        <b>Google</b>
    </a>
    <a class="list-group-item"
href="http://www.stackoverflow.com">
        <b>Stackoverflow</b>
    </a>
    <a class="list-group-item"
href="http://www.w3schools.com">
        <b>W3Schools</b>
    </a>
</div>
```

and the visual result will actually look like this:

Great; exactly what we wanted. But how did that work?

Let's look at the hello.html template again quickly:

```
<div layout:fragment="content">
    <div class="list-group">
        <a th:each="resource : ${resources}"
           th:href="${resource.url}" class="list-group-
item">
            <b th:text="${resource.name}"></b>
        </a>
    </div>
</div>
```

The "list-group" and "list-group-item" classes are just Twitter Bootstrap classes that give us the look and feel that we see above.

The real Thymeleaf magic comes from *th:each*. This loops through each item in the "*resources*" model attribute (works with any java *Iterable*) creating a Thymeleaf variable called "resource" to represent each item.

We then see the use of another important Thymeleaf attribute *th:href*. This is used to set the href attribute on the given anchor element (…). Note that there are matching Thymeleaf attributes for pretty much every possible HTML tag attribute that you would want to set (http://www.thymeleaf.org/doc/tutorials/2.1/usingthymeleaf.html#setting-value-to-specific-attributes).

The final thing to notice is the use of Thymeleaf's OGNL (Object Graph Notation Language). It is basically a dot-notation syntax that allows access to properties within java objects (via getters). Here you can see we have used ${resource.url} to access the getUrl() property of our DeveloperResource object and ${resource.name} to access getName(). Note: this also works to multiple levels of nesting (e.g. ${**resource.name.class.simpleName**}) as long as the fields are all accessible via getters.

Conditionals

What if we want a certain HTML element/block to only be rendered under certain conditions?

Let's extend our existing *hello.html* template to include such a scenario.

Add, to the beginning of our content block, the following line:

```
<span th:if="${resources.size() == 0}">No resources found.
</span>
```

Suddenly we have a useful informational message that appears if our *"resources"* model attribute is empty.

If, and only if, the resources array has size 0, the "No resources found" *span* element will be rendered in the HTML.

Forms

We often want to use forms to submit data to our server. In order to re-use templates for different scenarios it can be very useful to be able to specify the *action* property of a form from the server.

You have probably already guessed how this is done:

```
<div layout:fragment="content">
    <form th:method="POST" th:action="${action}">
        <div class="form-group">
            <label>Name:</label>
            <input class="form-control" name="name">
        </div>
        <button type="submit" class="btn btn-
default">Submit</button>
    </form>
</div>
```

As expected, we just use a *th:action* attribute instead of a normal *action* attribute.

What you'll also notice, however, is that we've used a *th:method* attribute to specify the form *method* too:

```
<div layout:fragment="content">
    <form th:method="POST" th:action="${action}">
        <div class="form-group">
            <label>Name:</label>
            <input class="form-control" name="name">
        </div>
        <button type="submit" class="btn btn-
default">Submit</button>
    </form>
</div>
```

Given that we're not using a model attribute for this field, why bother using the Thymeleaf syntax?

Well, Thymeleaf also automatically provides the very useful functionality of adding CSRF tokens to our forms out of the box; but **only** if we set the *action* and *method* attributes using *th:action* and *th:method* respectively.

This is important because, if you are using Spring Security, then your server will be CSRF protected (as it should), and any POST/PUT/DELETE requests will automatically fail; unless they come with a CSRF token.

Simply by setting the *method* and *action* attributes on your form using Thymeleaf (th:method and th:action), you will automatically get CSRF tokens injected into your forms.

The resulting rendered html for the above form will look as follows:

```
<form method="POST" action="/myaction">
    <div class="form-group">
        <label>Name:</label>
        <input class="form-control" name="name" />
    </div>
    <button type="submit" class="btn btn-
default">Submit</button>
    <input type="hidden" name="_csrf" value="e26856cf-7535-
42e9-9afa-49b06aa6fa71" />
</form>
```

As you can see, Thymeleaf has automatically added the CSRF token to the bottom of the form for you.

Nice.

Summary

There is a lot more to Thymeleaf than this (covered in great detail here) but there should be enough here to get you started.

In summary:

- Adding Thymeleaf to your Spring Boot MVC project is easy:

```
compile('org.springframework.boot:spring-boot-starter-
thymeleaf:1.3.1.RELEASE')
```

- Remember to turn off the Thymeleaf cache during development:

```
spring.thymeleaf.cache=false
```

- **layout:decorator** and **layout:fragment** can be used for code re-use in all your templates.
- **th:text** and **th:utext** *replaces* the content of HTML tags with server-side content.
- **${name}** and **|Hello ${name}|** inject model parameters and String-interpolated model parameters into your templates, respectively.
- **th:each** will loop over Java iterables and generate content for each item.
- Thymeleaf provides custom attributes (e.g. **th:href**) for almost every standard HTML attribute (more details here)
- Thymeleaf's OGNL lets you access nested properties of your model parameters (e.g. ${**resource.name.class.simpleName**}) via getters.
- **th:if** will only render a given HTML element if the value of its expression is true.
- And **always** use **th:method** and **th:action** when creating forms; so that you get CSRF protection out of the box.

Any questions/comments/feedback, very welcome.