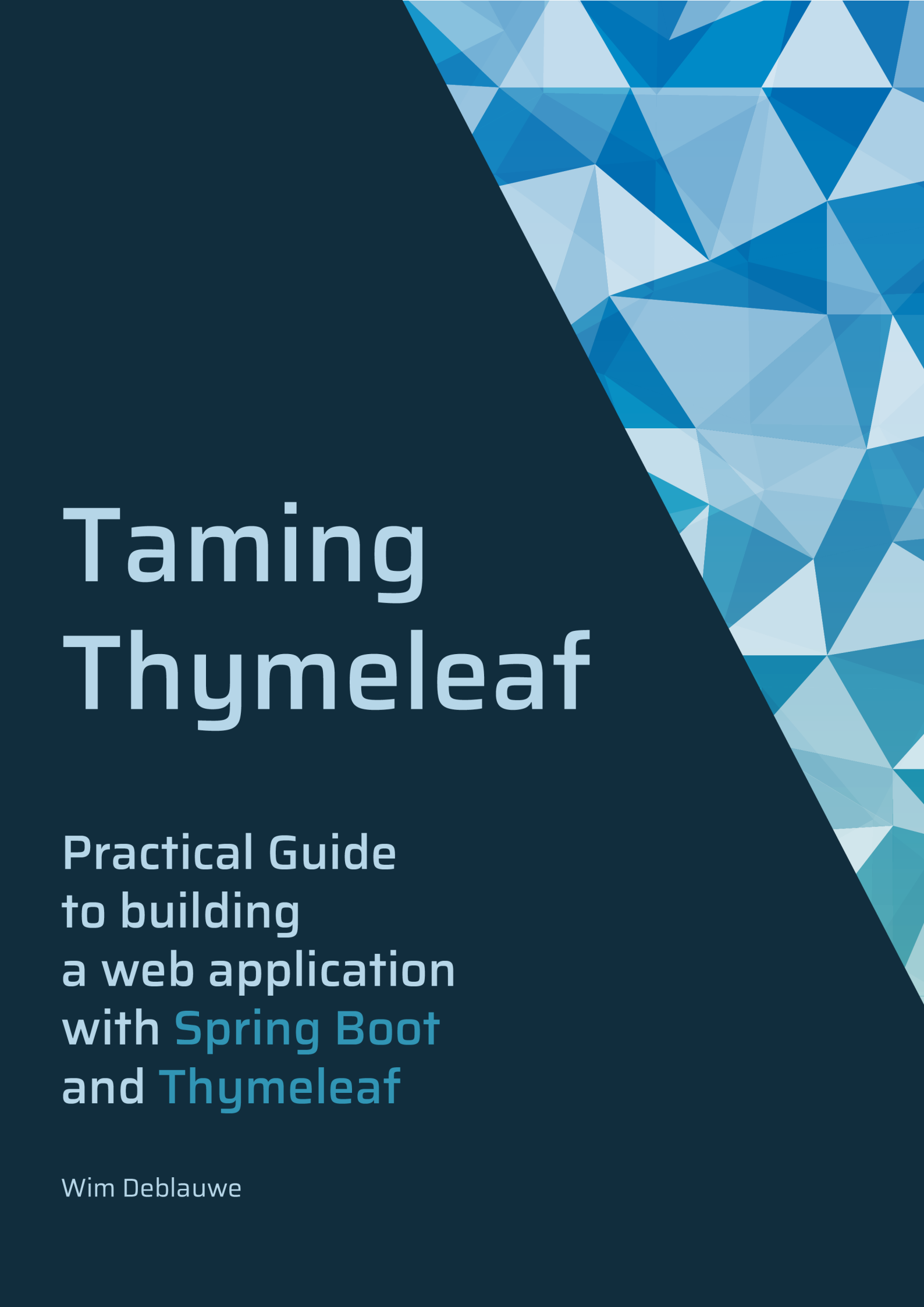


Taming Thymeleaf

Practical Guide to building a web application with Spring Boot and Thymeleaf

Wim Deblauwe

Taming Thymeleaf (Sample)

Practical Guide to building a web application with Spring Boot and Thymeleaf

Wim Deblauwe

Version 1.0.0-SNAPSHOT, 2020-12-26

Table of Contents

Sample introduction

This book is an excerpt from the Taming Thymeleaf book by Wim Deblauwe.

See the full table of contents and get the book at https://leanpub.com/taming-thymeleaf

Find Wim at @wimdeblauwe on Twitter or email him at wim.deblauwe@gmail.com if you have any questions.

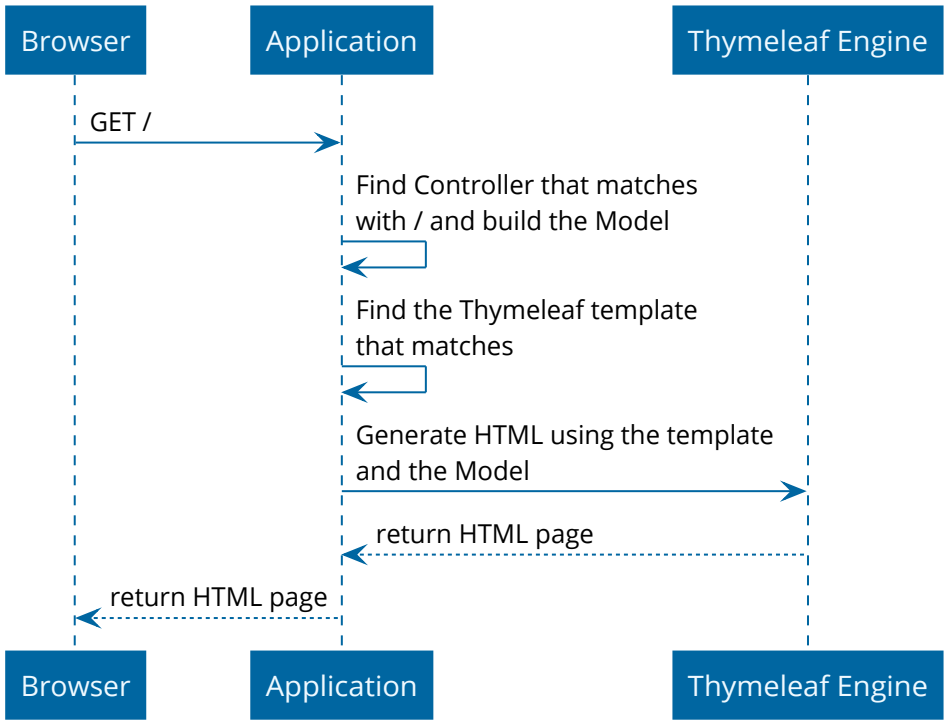
Chapter 1. Thymeleaf introduction

1.1. What is Thymeleaf?

Thymeleaf is a server-side Java template engine that can be used in web and standalone environments. It is mainly used for HTML, but can also be used for XML, JavaScript, CSS or plain text.

Thymeleaf templates are plain HTML files and are stored in the `src/main/resources/templates` folder in a Spring Boot application. [1]

This diagram shows how server-side rendering works:



1. The browser starts by doing a GET request over the network to the server where the application runs.
2. The application will match the requested path of the URL to a Controller. This is a piece of software in our application that will build a kind of Map of java objects that will be used by the template during rendering. We call this map the *Model*.
3. The application finds the Thymeleaf template to use for rendering.
4. The application uses the Thymeleaf engine (also running inside the application) to combine the template with the Java objects in the model. This results in an HTML page.
5. The application returns the generated HTML page to the browser where the browser renders it.

1.2. Writing our first template

Let's dive right in, building our first template. Starting from the Spring Boot project we created, we add a new HTML page at `src/main/resources/templates` called `index.html`:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Taming Thymeleaf</title>
</head>
<body>
<h1>Taming Thymeleaf</h1>
<div>This is a thymeleaf page</div>
</body>
</html>
```

Start the Spring Boot application and open http://localhost:8080 in your browser. The result should be similar to this:

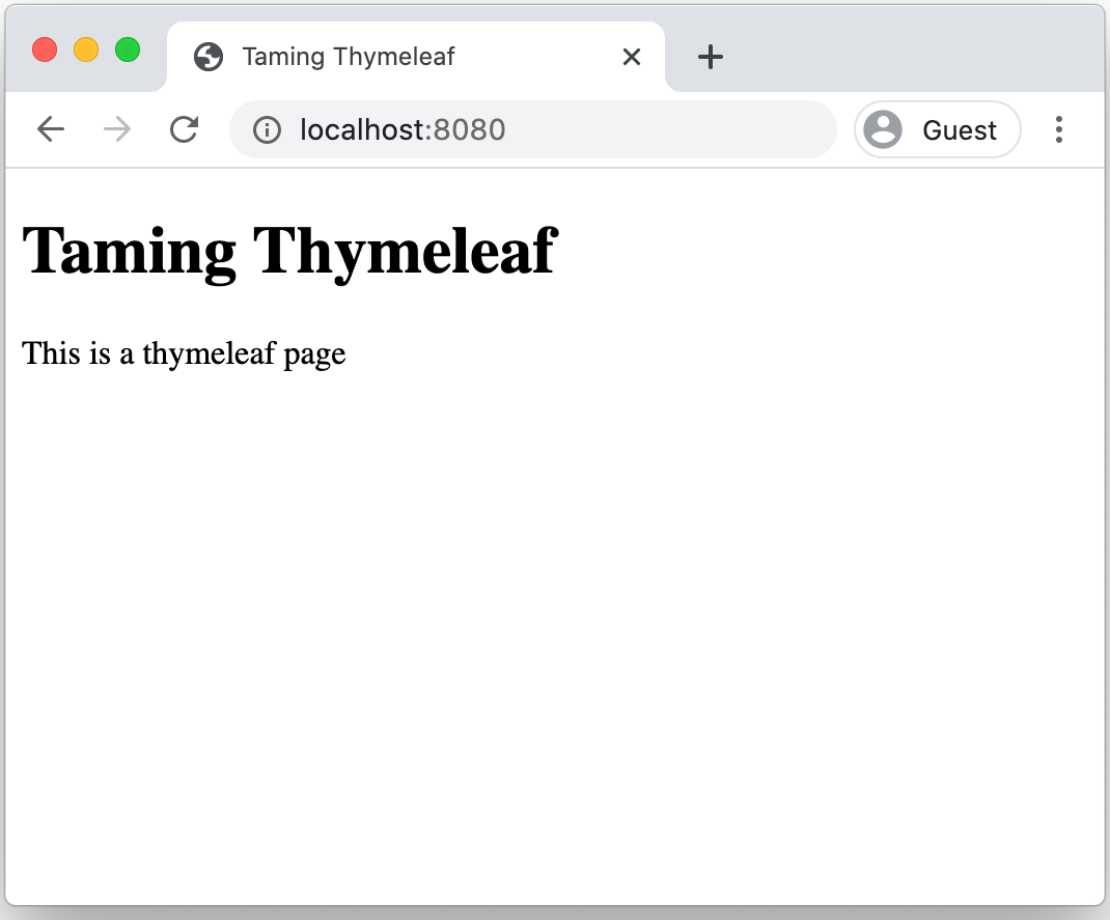


Figure 1. Thymeleaf rendering plain HTML page

Of course, there is currently nothing on the page where Thymeleaf actually has to do something. Let's put the template engine to work.

Update the `index.html` page to this:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org" ①
      lang="en">
<head>
    <meta charset="UTF-8">
    <title>Taming Thymeleaf</title>
</head>
<body>
<h1>Taming Thymeleaf</h1>
<div th:text="|Sum of 2 + 2 = ${ 2 + 2 }|"></div> ②
</body>
</html>
```

① Thymeleaf `th:` namespace declaration. Thymeleaf adds custom tags and attributes to HTML. To avoid naming conflicts and for clarity, those tags and attributes are put in an XML namespace.

② Use a Thymeleaf expression via the `th:text` attribute. Thymeleaf will first evaluate the expression inside the attribute and put the result as the body of the `<div>` tag that contains the `th:text` attribute.

Restart the application and refresh the browser:

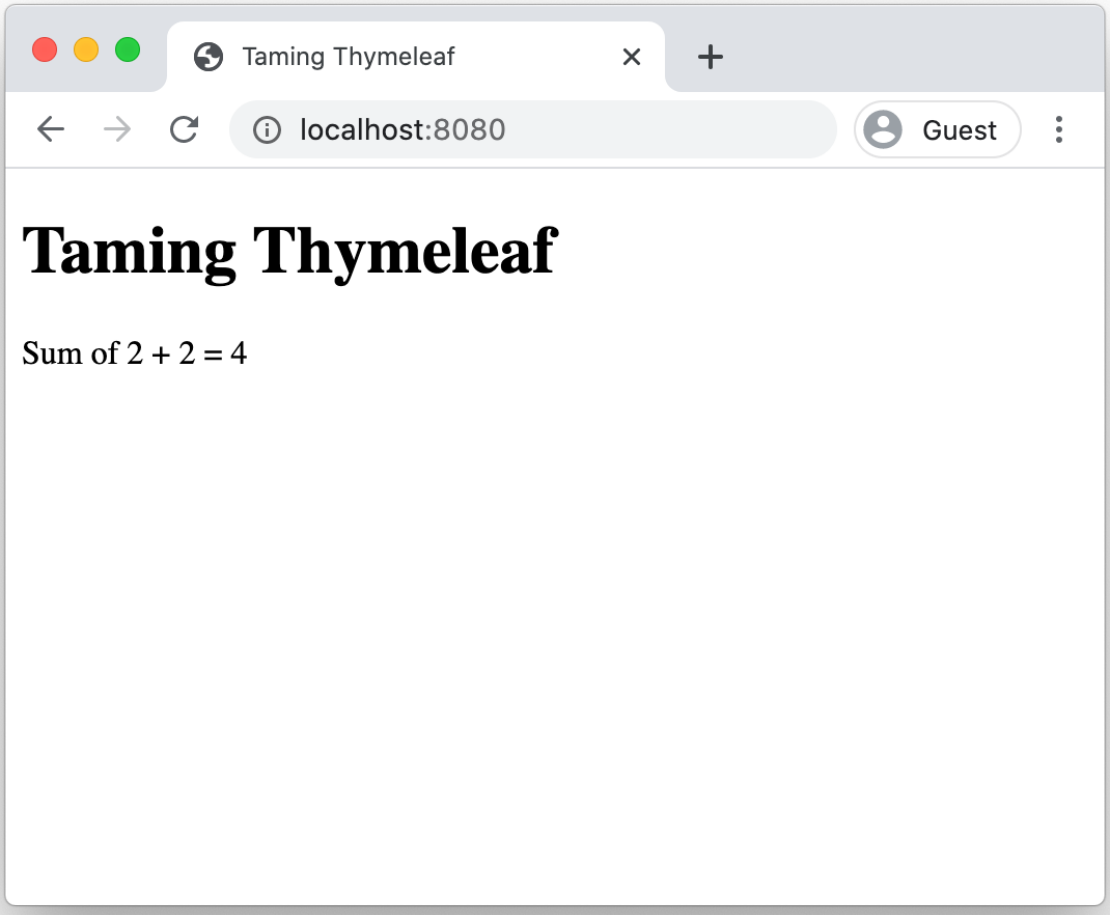


Figure 2. Thymeleaf rendering summation expression

Use the 'View source' functionality of your browser to see the exact HTML that Thymeleaf has rendered.

Natural templates

Thymeleaf uses *natural templates*. These are HTML files where the basic structure is still normal HTML tags, but where the dynamic behaviour is defined by Thymeleaf attributes.

The advantage here is that the styling could be done outside of the running application, which might be easier for a designer.

We will set up a live-reload system for the running application later so we don't *need* to use static templates for styling, while still having a very fast development cycle.

1.3. Writing our first controller

Our template so far was nice, but not very useful. We obviously want the page to render stuff from

our application. Let's do that next.

We are using Thymeleaf with Spring MVC. *MVC* is an acronym for Model View Controller. It is a well-known design pattern. You can find more information about it at Model–view–controller on Wikipedia.

There are 3 parts in the pattern:

- The *View* part is what we have already done when we created our `index.html` template. It is the visual part of the design pattern.
- The *Controller* is what we are going to create next. The controller is responsible for fetching the data from the application, and showing the correct view according to the requested URL. It usually has no actual business logic, but delegates to other components in the application.
- The *Model* is the object that the controller passes to the view with the data to be used for the rendering of the template.

A controller in Spring MVC is a simple Java class that is annotated with `@Controller`:

```
package com.tamingthymeleaf.application;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;

import java.util.List;

@Controller ①
@RequestMapping("/") ②
public class RootController {

    @GetMapping ③
    public String index(Model model) { ④
        model.addAttribute("pageTitle", "Taming Thymeleaf"); ⑤
        model.addAttribute("scientists", List.of("Albert Einstein",
                                                 "Niels Bohr",
                                                 "James Clerk Maxwell")); ⑥

        return "index"; ⑦
    }
}
```

① The `@Controller` annotation indicates to Spring Boot that this class is a controller. The component scanning will automatically create an instance of this class and add it to the Spring Context.

② The `@RequestMapping` annotation sets the root path of the URL for all methods of the class.

③ `GetMapping` indicates that an HTTP GET will call this method. Note that the name of the method (`index` in the example) really does not matter at all for the working of the application.

④ Controller methods can declare parameters of certain types that Spring MVC will inject with the proper instances. We will later see some other examples, but `Model` is one of the most important ones. It allows adding attributes that Thymeleaf can use to render the data.

⑤ We add a simple `String` value under the `pageTitle` key to the model.

⑥ We can also add complex objects or collections to the model.

⑦ The return value of a controller method has a few options. One of the simplest ones is to return a `String`. The value will be interpreted as the path to the template. With default Spring Boot, this is relative to the `src/main/resources/templates` directory.

Component scanning

Spring heavily utilizes something called *component scanning*. At startup, Spring will search for classes on the classpath that are annotated with certain annotations like `@Component`, `@Configuration`, `@Controller`, `@Service`, `@Repository`, ...

When it finds those, it will automatically register them in the context by creating an instance, and injecting any declared dependencies (in the constructor usually).

See https://www.baeldung.com/spring-component-scanning for more details.

With this controller in place, we can update the template:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org"
      lang="en">
<head>
    <meta charset="UTF-8">
    <title>Taming Thymeleaf</title>
</head>
<body>
<h1 th:text="${pageTitle}">Taming Thymeleaf</h1> ①
<div>
    <ul>
        <li th:each="scientist : ${scientists}"> ②
            <span th:text="${scientist}"></span>
        </li>
    </ul>
</div>
</body>
</html>
```

① Use the `pageTitle` model attribute. Note that the actual text `Taming Thymeleaf` inside the `<h1>` tag does not matter at all. Thymeleaf will overwrite it with the contents of `pageTitle`.

② Use the `th:each` tag to loop over our collection of scientists.

Start the application and refresh the browser:

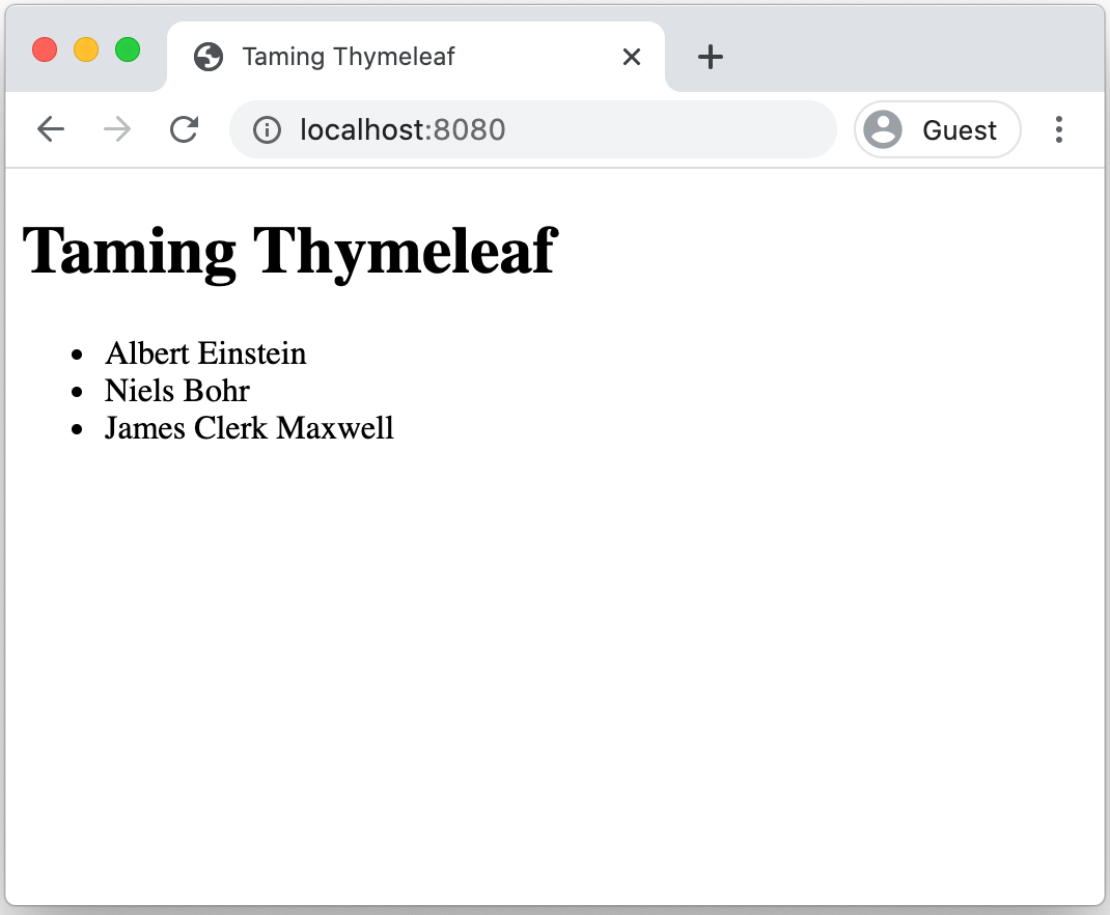


Figure 3. Using a controller to supply data to the view

1.4. Thymeleaf expressions

The expression inside the `th:*` HTML attributes are OGNL (Object-Graph Navigation Language) expression by default. However, in a Spring Boot application, those are SpringEL (Spring Expression Language) expressions. Luckely, for most cases, the syntax is exactly the same.

1.4.1. Variables

When a Thymeleaf template gets processed, the application will put variables in the context via the controller. Those variables can be referenced in the templates via the `${…}` syntax.

For example:

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

Suppose there is a `String` in the Thymeleaf context with the name `username` that has the value `John Doe`, then the HTML will be rendered as:

```
<div>John Doe</div>
```

The variable in the context does not need to be a `String`. Other types will have their `toString()` method invoked.

The Thymeleaf variable syntax is not limited to the exact object that is placed on the context. We can call methods:

```
<div th:text="${user.getName()}"></div>
```

Or if the method name adheres to the JavaBean specification, we can simulate property access:

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

Map

If the object in the context is a `Map`, then dot notation can be used to access a value via its key:

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

Alternatively, use the bracket syntax. For certain keys (E.g. if they contain spaces) you will need to use the bracket syntax:

```
<div th:text="${capitalsOfTheWorld['The Netherlands']}"></div>
```

Array or List

Collections that allow indexed access can be used like this:

```
<div th:text="${vehiclesList[0].name}"></div>
```

1.4.2. Text

Most applications will require that they can be translated into the language of the user. Even if it is not a requirement at the start, you probably want to do this in case it ever becomes a requirement.

The syntax for this is:

```
<h1 th:text="#{dashboard.title}"></h1>
```

By default, Spring Boot will pick up translations from a `src/main/resources/messages.properties` file.

src/main/resources/messages.properties

```
dashboard.title=Dashboard
```

Additional languages can be added by adding another such file, but postfixing the file name with the locale. For example, use `messages_nl.properties` for Dutch translations.

1.4.3. Selected objects

Thymeleaf has a kind of shortcut syntax in case you need to use many properties of a single object. Suppose you have a template that displays information about a car like this:

```
<div>
    <p>Brand: <span th:text="${car.brand}"></span></p>
    <p>Type: <span th:text="${car.type}"></span></p>
    <p>Fuel: <span th:text="${car.fuelType}"></span></p>
    <p>Color: <span th:text="${car.color}"></span></p>
</div>
```

You can avoid the duplication of the `car` variable by *selecting* the variable with the `th:object` attribute and refer to the properties of the selected object using the `*{…}` syntax:

```
<div th:object="${car}">
    <p>Brand: <span th:text="*{brand}"></span></p>
    <p>Type: <span th:text="*{type}"></span></p>
    <p>Fuel: <span th:text="*{fuelType}"></span></p>
    <p>Color: <span th:text="*{color}"></span></p>
</div>
```

If there is no object selected, then `${…}` and `*{…}` are equivalent.

1.4.4. Link to URLs

What would a web application be without URLs? It would have be very simple I guess.

As URLs are so important, there is a special syntax for them: `@{…}`

This can be used for an absolute URL:

```
<a th:href="@{https://www.google.com/search?q=thymeleaf"></a>
```

or a relative URL:

```
<a th:href="@{/users}"></a>
```

We can also use variables inside those links.

This is equivalent to the first example, given `searchTerm` is a context variable that contains the `thymeleaf` string.

```
<a th:href="@{https://www.google.com/search(q=${searchTerm})"></a>
```

If the variable is *not* referenced in the URL itself, it is added as a query parameter. If it is referenced, it can be used as a path variable:

```
<!-- Will output '/users/123/edit'  -->
<a th:href="@{/users/{userId}/edit(userId=${user.id})"></a>
```

1.4.5. Literal substitutions

If you need to combine a string literal with a variable, you can do something like this:

```
<div th:id="'container-' + ${index}"></div>
```

Thymeleaf has a shortcut syntax that is equivalent using the pipe (`|`) symbol:

```
<div th:id="|container-${index}|"></div>
```

1.4.6. Expression inlining

Instead of using `th:*` tags to use variables, it might be desirable at times to directly put a variable result in HTML. This is possible in Thymeleaf using expression inlining.

For example:

```
<span>The total price is [[${totalPrice}]]</span>
```

This will render to:

```
<span>The total price is € 5.00</span>
```

This is equivalent of:

```
<span>The total price is <span th:text="${totalPrice}">€ 19.99</span></span>
```

The final rendering in that case (Given `totalPrice` is `€ 5.00`):

```
<span>The total price is <span>€ 5.00</span></span>
```

1.5. Thymeleaf attributes

The Thymeleaf expressions we just saw can be used inside Thymeleaf attributes. For example, the `th:text` attribute is one of those. This section will show the most important ones.

1.5.1. Element text content

`th:text` will place the result of the expression inside the tag it is declared on.

For example:

```
<div th:text="${username}">Bob</div>
```

Will render as:

```
<div>Jane</div>
```

Given the `username` variable in the context contains `Jane`.

1.5.2. Element id attribute

`th:id` will add an `id` attribute with the result of the expression on the tag it is declared on.

For example:

```
<div th:id="|container-${userId}|"></div>
```

Will render as:

```
<div id="container-1"></div>
```

Given the `userId` variable in the context contains `1`.

1.5.3. Conditional inclusion

`th:if` will render the tag it is declared on only if the expression evaluates to true.

For example:

```
<div th:if="${user.followerCount > 10}">You are famous</div>
```

Will render as:

```
<div>You are famous</div>
```

Given the `user.followerCount` variable in the context a value greater than `10`. If the variable is less than `10`, the `<div>` will not be rendered in the output.

1.5.4. Conditional exclusion

`th:unless` will render the tag it is declared on only if the expression evaluates to `false`.

For example:

```
<div th:unless="${user.followerCount > 0}">You have no followers currently.</div>
```

Will render as:

```
<div>You have no followers currently</div>
```

Given the `user.followerCount` variable in the context is exactly `0`. If the variable is greater than `0`, the `<div>` will not be rendered in the output.

Thymeleaf has no `if`/`else` statement, but this can be easily done by combining `th:if` with `th:unless`. For example:

```
<div th:if="${user.followerCount > 0}">You have <span
th:text="${user.followerCount}"></span> followers currently.</div>
<div th:unless="${user.followerCount > 0}">You have no followers
currently.</div>
```

Either the first `<div>` or the 2nd one will be rendered depending on the value of `user.followerCount`.

1.5.5. Iteration

`th:each` allows iterating over a collection. It will create as many tags as there are items in the collection.

For example:

```
<ul>
    <li th:each="scientist : ${scientists}" th:text="${scientist.name}"></li>
</ul>
```

Will render as:

```
<ul>
    <li>Marie Curie</li>
    <li>Erwin Schrödinger</li>
    <li>Max Planck</li>
</ul>
```

Given the `scientists` variable in the context references a collection of objects that have a `name` property.

Thymeleaf will do the "right thing" you expect in the above example and first evaluate the `th:each` and then the `th:text`. There is a defined precedence in the attribute processing that ensures the proper order. See Attribute Precedence on the Thymeleaf website for the exact details.

1.6. Preprocessing

Thymeleaf has a preprocessing expression. This allows to first execute the preprocessing and use the result of that in the final rendering of the templates.

This will be especially useful for [Fragments] which we will cover later.

As a simple example, consider this snippet:

```
<h1 th:text="#{__${title}__}"></h1>
```

Thymeleaf will first substitute `${title}` with the value of that parameter. Assume `users.title` for example.

This will turn the template into the following:

```
<h1 th:text="#{users.title}"></h1>
```

In a 2nd step, Thymeleaf will now execute the `th:text` and search for the translation key (Due to `#{…}`) of `users.title` and display that in the `<h1>`.

Final result:

```
<h1>Users</h1>
```

1.7. Summary

In this chapter, you learned:

- Thymeleaf templating basics including expressions and attributes.
- Using a controller to pass data from the application to the view.

[1] If you want to use another directory,see Changing the Thymeleaf Template Directory in Spring Boot for more info on how to do that.

Closing

I hope you enjoyed this free sample of the Taming Thymleaf book.

Keep your learning going by getting the full book at https://leanpub.com/taming-thymeleaf

Happy coding!