

# Taming Thymeleaf

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

Wim Deblauwe

Ещё больше книг по Java в нашем телеграм канале:  
<https://t.me/javalib>

# Taming Thymeleaf

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

**Wim Deblauwe**

Version 2.0.2, 2022-10-19

# Table of Contents

Taming Thymeleaf .....	1
Dedication .....	2
Acknowledgements .....	3
Introduction .....	4
Source code .....	5
1. What are Spring Boot and Thymeleaf? .....	6
1.1. Spring Framework .....	6
1.2. Spring Boot .....	6
1.3. Thymeleaf .....	7
2. Getting started .....	8
2.1. Prerequisites .....	8
2.1.1. macOS/Linux .....	8
2.1.2. Windows .....	8
2.2. Spring Initializer .....	9
2.3. Summary .....	14
3. Thymeleaf introduction .....	15
3.1. What is Thymeleaf? .....	15
3.2. Writing our first template .....	15
3.3. Writing our first controller .....	18
3.4. Thymeleaf expressions .....	21
3.4.1. Variables .....	21
3.4.2. Text .....	22
3.4.3. Selected objects .....	23
3.4.4. Link to URLs .....	23
3.4.5. Literal substitutions .....	24
3.4.6. Expression inlining .....	24
3.5. Thymeleaf attributes .....	25
3.5.1. Element text content .....	25
3.5.2. Element id attribute .....	25
3.5.3. Conditional inclusion .....	25
3.5.4. Conditional exclusion .....	26
3.5.5. Iteration .....	26
3.6. Preprocessing .....	27
3.7. Summary .....	28
4. Thyme Wizards .....	29
4.1. Cascading Style Sheets .....	29
4.2. Tailwind CSS .....	30
4.2.1. What is Tailwind CSS? .....	30
4.2.2. Adding Tailwind CSS .....	32
4.2.3. Configure Maven .....	37
4.2.4. Live reload .....	40
4.2.5. Tailwind CSS design system configuration .....	43
4.3. Application shell .....	45
4.3.1. Tailwind UI .....	45
4.3.2. Client side interactivity .....	48

4.3.3. Serving static images .....	60
4.4. Summary .....	62
5. Fragments .....	63
5.1. What are fragments? .....	63
5.2. Using fragments .....	63
5.3. Fragments with parameters .....	65
5.4. Fragments with HTML snippets as arguments .....	66
5.5. Inline separate SVG files .....	68
5.6. Homepage refactoring .....	70
5.7. Menu item components .....	74
5.8. Summary .....	82
6. Layouts .....	83
6.1. What is the Thymeleaf Layout Dialect? .....	83
6.2. Layouts with parameters .....	86
6.3. Page titles .....	88
6.4. Homepage refactoring .....	89
6.5. Summary .....	92
7. Controllers .....	93
7.1. What is a controller? .....	93
7.2. Exposing data to the view .....	94
7.3. Path parameters .....	96
7.4. Posting data .....	98
7.5. Support for other HTTP methods .....	98
7.6. Team and User controllers .....	99
7.7. Summary .....	106
8. Internationalization .....	107
8.1. Internationalization basics .....	107
8.2. Using a query parameter to select the language .....	108
8.3. Menu items translations .....	110
8.4. Summary .....	113
9. Database connection .....	114
9.1. Spring Data JPA .....	114
9.2. PostgreSQL database .....	114
9.3. Getting started with Spring Data JPA .....	115
9.3.1. Add Spring Data JPA to the project .....	115
9.3.2. User Entity .....	117
9.3.3. User repository .....	120
9.3.4. User Repository Test .....	122
9.3.5. Adding properties to <code>User</code> .....	129
9.4. Summary .....	134
10. Displaying data .....	135
10.1. Generate random users .....	135
10.2. Get users on the HTML page .....	140
10.3. Refactor the table using fragments .....	144
10.4. Use pagination .....	147
10.5. Hide columns on mobile .....	160
10.6. Summary .....	162
11. Forms .....	163

11.1. Form fields . . . . .	163
11.2. Error messages . . . . .	174
11.3. Custom error messages . . . . .	179
11.4. Custom validator . . . . .	182
11.5. Errors summary . . . . .	187
11.6. Validation groups and order . . . . .	190
11.7. Summary . . . . .	194
12. Data editing . . . . .	195
12.1. Add user button . . . . .	195
12.2. Edit user data . . . . .	197
12.3. Refactoring to fragments . . . . .	210
12.4. Handling Optimistic Locking failure . . . . .	212
12.5. Custom error pages . . . . .	216
12.6. Summary . . . . .	221
13. Implement deletion of an entity . . . . .	222
13.1. Using a dedicated URL . . . . .	222
13.2. Using the DELETE HTTP method . . . . .	232
13.3. Flash attributes . . . . .	234
13.4. Summary . . . . .	238
14. Security . . . . .	239
14.1. Default Spring Security . . . . .	239
14.2. Hardcoded password . . . . .	243
14.3. User roles . . . . .	245
14.3.1. URL based authorization . . . . .	245
14.3.2. Annotation based authorization . . . . .	250
14.4. Thymeleaf integration . . . . .	251
14.4.1. User specific views . . . . .	251
14.4.2. Current logged on user information . . . . .	253
14.5. Custom logon page . . . . .	255
14.6. Users from database . . . . .	264
14.6.1. User entity updates . . . . .	264
14.6.2. Spring Security connection . . . . .	269
14.6.3. Show current user info . . . . .	275
14.6.4. Create user form . . . . .	276
14.6.5. Refactor the edit user implementation . . . . .	282
14.7. Summary . . . . .	289
15. Testing . . . . .	290
15.1. Using @WebMvcTest . . . . .	290
15.1.1. Getting started with @WebMvcTest . . . . .	291
15.1.2. Authenticating in the @WebMvcTest . . . . .	294
15.1.3. Using HtmlUnit . . . . .	298
15.2. Using Cypress . . . . .	305
15.2.1. Cypress installation . . . . .	305
15.2.2. First Cypress test . . . . .	306
15.2.3. Bypassing login . . . . .	313
15.2.4. Running Cypress tests from JUnit . . . . .	315
15.2.5. JUnit Testfactory . . . . .	321
15.3. Summary . . . . .	325

16. Various tips and tricks .....	326
16.1. Open Session In View.....	326
16.1.1. What is it? .....	326
16.1.2. Consequences of disabling .....	327
16.2. StringTrimmerEditor .....	327
16.3. Global model attributes.....	329
16.3.1. Controller specific .....	329
16.3.2. Application wide .....	330
16.4. File upload .....	332
16.5. Selecting a linked entity value.....	342
16.5.1. Implementation.....	342
16.5.2. Tests .....	359
16.6. Dynamically adding rows.....	362
16.6.1. Entities .....	363
16.6.2. Static server side rendering.....	367
16.6.3. Make updates persistent.....	376
16.6.4. Add rows.....	381
16.6.5. Delete rows .....	385
16.7. Custom editors and formatters .....	389
16.7.1. Custom editor .....	389
16.7.2. Custom formatter .....	391
16.8. Date picker.....	393
16.8.1. Duet Date Picker .....	394
16.8.2. Internationalization of date picker .....	399
17. Closing .....	402
Appendix A: Change log.....	403

# Taming Thymeleaf

© 2022 Wim Deblauwe. All rights reserved. Version 2.0.2.

Published by Wim Deblauwe ([wim.deblauwe@gmail.com](mailto:wim.deblauwe@gmail.com))

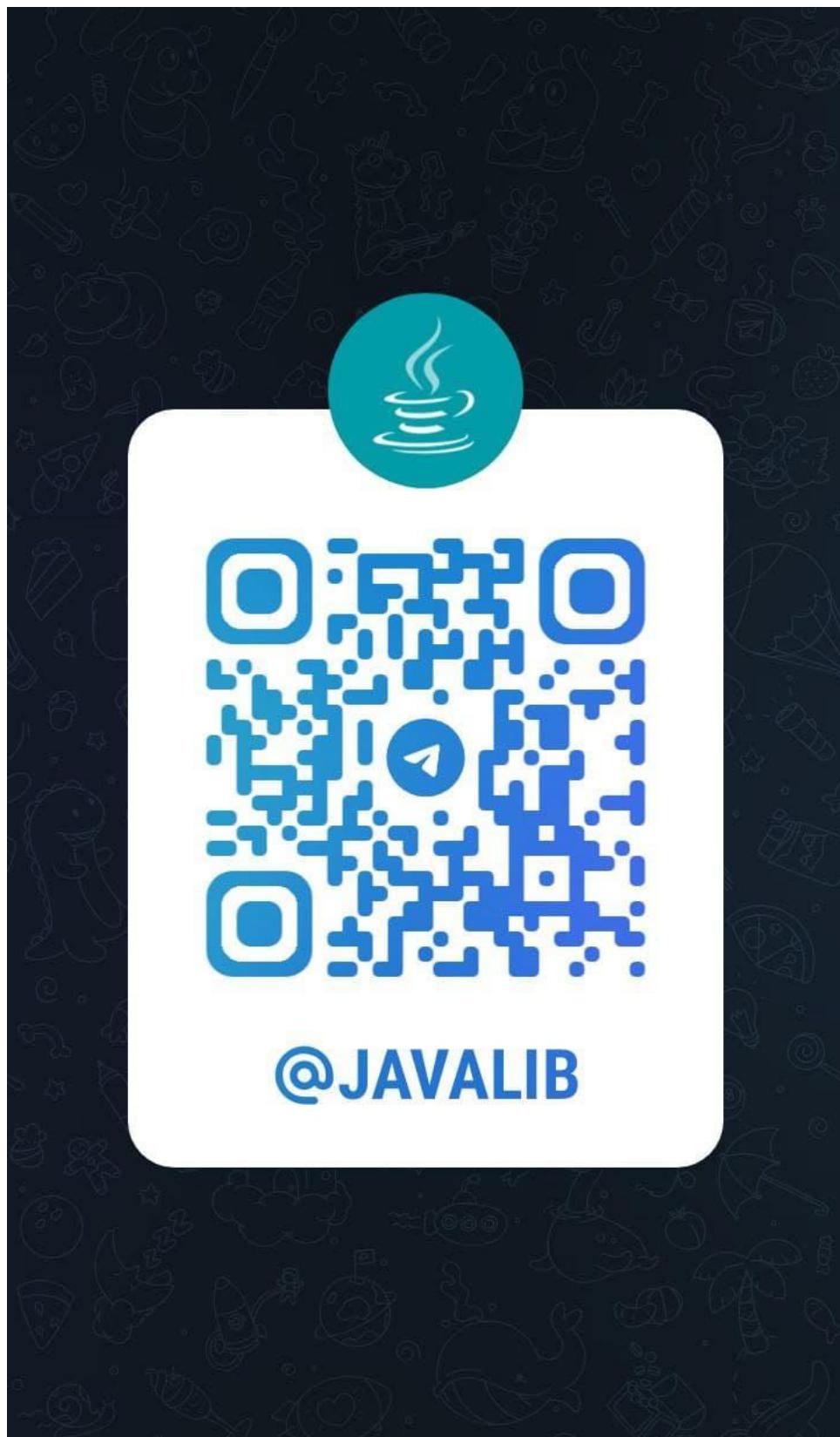
No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recoding, scanning or otherwise except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without the prior written permission of the publisher.

While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors and omissions, or for any damage resulting from the use of the information contained herein. The book solely reflects the author's views.

Cover Design: Jasmine Verhaeghe

# Dedication

I would like to dedicate this book to my wife Sofie and sons Victor and Jules. Their continued support for all my endeavours means the world to me.



# Acknowledgements

I would like to thank all the people that made Spring and Spring Boot a reality. It is really an amazing piece of software.

I would also like to thank everybody that created and/or contributed to Thymeleaf. I remain convinced that it is still one of the best ways of creating an HTML frontend for many use-cases.

I also want to send a big thank you to the people that created Asciidoctor and to [Alexander Schwartz](#) for his amazing work on the [IntelliJ Asciidoc plugin](#). It made writing this book extremely enjoyable.

Further, I also want to thank my sister-in-law Jasmine Verhaeghe for the work on the book's cover. I am really happy with how it looks.

Finally, I want to thank [Philip Riecks](#) for reviewing the book. His feedback has been invaluable for making this book the best it can be.

# Introduction

I have been working with Spring Boot for over four years and it has made development a tremendous joy. It is important in our fast-paced world to be able to prototype quickly, but also to ensure that you are not doing any wasted work.

For me, this is one of the major strengths of Spring Boot. The smallest application can fit in a [tweet](#), yet your application will scale to Internet scale with the greatest of ease.

Combine Spring Boot with Spring Data and Spring Security and you can have something up and running in no time. And it is not just "something", it is a solid base to build upon.

For the "front" of a web application, there are 2 categories: so called "traditional" server-side rendered HTML, or Single Page Applications (SPA). Thymeleaf uses server-side rendered HTML which is still a very valid implementation choice today, as [confirmed by many of the people that Marco Behler interviewed for his blog](#). JavaScript-heavy SPA's surely also have their place, but in most cases they are overkill.

This book is the culmination of four years of working with Spring Boot on a variety of projects. This is the book I wished I had when starting out building back-end applications with Java, Spring Boot and Thymeleaf. It will give you all the basics you need to start developing an application using Thymeleaf and Spring Boot.

Through the creation of an application for a fictional basketball team called [Thyme Wizards](#), you will learn about Spring, Spring Boot, Spring Security, Spring Data and Thymeleaf. You will also learn to use unit and integration tests to ensure the proper code functionality and build a maintainable code base that you can expand upon.

This book assumes you have enough basic Java knowledge to be able to follow. However, proficiency in similar languages like C# should be sufficient.

Thank you so much for buying this book. I really hope it helps you in your journey to learn Spring Boot and Thymeleaf. Please let me know if it did via [@wimdeblauwe](#) on Twitter or email me at [wim.deblauwe@gmail.com](mailto:wim.deblauwe@gmail.com). Thanks again!

# Source code

The source code of the book can be found on GitHub at <https://github.com/wimdeblauwe/taming-thymeleaf-sources>

There is a directory per chapter with how the code is suppose to look like by the end of each chapter.

# Chapter 1. What are Spring Boot and Thymeleaf?

## 1.1. Spring Framework

Spring Boot is based upon the [Spring Framework](#), which is at its core a dependency-injection container. Spring makes it easy to define everything in your application as loosely coupled components which Spring will tie together at run time. Spring also has a programming model that allows you to make abstractions from specific deployment environments.

One of the key things you need to understand is that Spring is based on the concept of "beans" or "components", which are basically singletons without the drawbacks of the traditional [singleton pattern](#).

With dependency injection, each component just declares the collaborators it needs, and Spring provides them at run time. The biggest advantage is that you can easily inject different instances for different deployment scenarios of your application (e.g., staging versus production versus unit tests).

The Spring portfolio includes a lot of sub-projects ranging from database access over security to cloud services.

The specific sub-projects used in this book are:

### **Spring Data**

Spring Data's mission is to provide a familiar and consistent, Spring-based programming model for data access while still retaining the special traits of the underlying data store.

### **Spring Security**

Spring Security is a powerful and highly customizable authentication and access-control framework. It is the de-facto standard for securing Spring-based applications.

### **Spring Web MVC**

Spring Web MVC is Spring's web framework built on the Servlet API.

*You can learn more about the core Spring Framework at [Spring Framework Documentation](#).*

## 1.2. Spring Boot

The Spring Boot website explains itself succinctly:

*Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run". We take an opinionated view of the Spring platform and third-party libraries so you can get started with minimum fuss. Most Spring Boot applications need very little Spring configuration.*

With Spring Boot, you get up and running with your Spring application in no time, without the need to deploy to a container like Tomcat or Jetty. You can just run the application right from your IDE.

Spring Boot also ensures that you get a list of versions of libraries inside and outside of the Spring portfolio that are guaranteed to work together without problems.

*You can learn more about Spring Boot from the excellent [Spring Boot Reference Documentation](#).*

## 1.3. Thymeleaf

Thymeleaf is a server-side Java template engine that uses natural templates to generate HTML pages. Natural templates are HTML templates that can correctly be displayed in browsers and work as static prototypes.

*Learn more about Thymeleaf at the [Thymeleaf Documentation](#).*



If you are familiar with PHP, you can think of Thymeleaf like [Blade templates](#).

# Chapter 2. Getting started

## 2.1. Prerequisites

To be able to create a Spring Boot application, we need to install Java and Maven or Gradle as a build tool.

In this book, we will be using Maven, but Gradle will work equally well.

We will use Java 17, which is the current LTS (Long Term Support) version of Java.

### 2.1.1. macOS/Linux

Use [SDKMAN!](#) to install Java and Maven.

1. Follow the SDKMAN! installation instructions at <https://sdkman.io/install>:

```
curl -s "https://get.sdkman.io" | bash
```

2. Install Java:

```
 sdk install java 17.0.1-tem
```



Use `sdk list java` to see a list of all possible Java versions that can be installed.

3. Install Maven:

```
 sdk install maven 3.8.4
```

4. Run `mvn --version` to see if both are configured correctly. The output should look similar to this:

```
wdb@Wims-MacBook-Pro ~ % mvn --version
Apache Maven 3.8.4 (9b656c72d54e5bacbed989b64718c159fe39b537)
Maven home: /Users/wdb/.sdkman/candidates/maven/current
Java version: 17.0.1, vendor: Eclipse Adoptium, runtime:
/Users/wdb/.sdkman/candidates/java/17.0.1-tem
Default locale: en_BE, platform encoding: UTF-8
OS name: "mac os x", version: "11.5", arch: "x86_64", family: "mac"
```

### 2.1.2. Windows

Use [Chocolatey](#) to install Java and Maven.

1. Follow the Chocolatey installation instructions at <https://chocolatey.org/install>.
2. Install Java

```
choco install temurin
```

3. Install Maven

```
choco install maven
```

4. Run `mvn -v` to see if both are configured correctly. The output should look similar to this:

```
Apache Maven 3.8.4 (9b656c72d54e5bacbed989b64718c159fe39b537)
Maven home: C:\ProgramData\chocolatey\lib\maven\apache-maven-3.8.4
Java version: 17.0.1, vendor: Eclipse Adoptium, runtime: C:\Program
Files\Eclipse Adoptium\jdk-17.0.1.12-hotspot
Default locale: nl_BE, platform encoding: Cp1252
OS name: "windows 10", version: "10.0", arch: "amd64", family:
>windows"
```

## 2.2. Spring Initializer

The easiest way to get started with Spring Boot is to create a project using Spring Initializr. This web application allows you to generate a Spring Boot project with the option of including all the dependencies you need.

To get started, open your favorite browser at <https://start.spring.io/>



**Project**

- Maven Project
- Gradle Project

**Language**

- Java
- Kotlin
- Groovy

**Spring Boot**

- 2.6.2 (SNAPSHOT)  2.6.1
- 2.5.8 (SNAPSHOT)  2.5.7

**Project Metadata**

Group	com.tamingthymeleaf		
Artifact	taming-thymeleaf		
Name	taming-thymeleaf		
Description	Example application for the Taming		
Package name	com.tamingthymeleaf.taming-thym		
Packaging	<input checked="" type="radio"/> Jar	<input type="radio"/> War	
Java	<input checked="" type="radio"/> 17	<input type="radio"/> 11	<input type="radio"/> 8

**Dependencies** ADD ... ⌘ + B

**Spring Web** WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

**Thymeleaf** TEMPLATE ENGINES

A modern server-side Java template engine for both web and standalone environments. Allows HTML to be correctly displayed in browsers and as static prototypes.

GENERATE ⌘ + ↵ EXPLORE CTRL + SPACE SHARE...

Figure 1. The Spring Initializr website.

Select the following options:

### Project

Maven project

### Language

Java

### Spring Boot

2.6.2

### Packaging

Jar [1]

### Java

17

### Dependencies

Select *Spring Web* and *Thymeleaf*



You can use Gradle as your build system or one of the other supported JVM languages if you prefer to. The main principles explained in the book remain the same.

Use the *Generate* button to download a zip file with the project, or use *Explore* if you want to see what would be generated.

The zip file contains the following files and directories:

```

├── .gitignore ①
├── .mvn ②
└── HELP.md ③
├── mvnw
└── mvnw.cmd
├── pom.xml ④
└── src
    ├── main
    │   ├── java
    │   │   └── com
    │   │       └── tamingthymeleaf
    │   │           └── application
    │   │               └── TamingThymeleafApplication.java ⑤
    │   └── resources
    │       ├── application.properties ⑥
    │       ├── static
    │       └── templates
    └── test
        └── java
            └── com
                └── tamingthymeleaf
                    └── application
                        └── TamingThymeleafApplicationTests.java ⑦

```

① A Git ignore file that has decent defaults for Maven, Spring Tool Suite, IntelliJ, Netbeans and Visual Studio Code

② The `.mvn` directory (with the `mvnw` executables) allows running Maven [without it being installed](#) on the system.

③ A help file that is generated with contents that is tailored to the dependencies that have been selected. It contains links to documentation of those dependencies.

④ Maven project `pom.xml` file that is configured with the selected dependencies and the [spring-boot-maven-plugin](#) to generate the standalone executable JAR file.

⑤ Main application file. The entry point of our Spring Boot application.

⑥ Properties file that allows to customize various parts of Spring Boot.

⑦ Integration test that will start the application to ensure the [Application Context](#) loads.

Build the application by running:

```
mvnw verify
```

This should output quite some information similar to this:

```
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.tamingthymeleaf:taming-thymeleaf-
application >-----
[INFO] Building Taming Thymeleaf 0.0.1-SNAPSHOT
[INFO] -----[ jar
]-----


...
[INFO]
-----
[INFO] BUILD SUCCESS ①
[INFO]
-----
[INFO] Total time: 5.341 s
[INFO] Finished at: 2021-12-11T20:16:43+01:00
[INFO]
```

① Verify that the build is successfully done.

Now that we have build the code, let's run it to see what it does. You can run the application from your IDE or through Maven with `mvn spring-boot:run`. See [Running your application](#) in the Spring Boot documentation for more information.

Running the application shows the following in the console:

```
.
   _-----
 / \ \ / _ _ ' _ _ _ _ ( _ ) _ _ _ _ _ \ \ \ \ \
( ( ) \ _ _ | ' _ | ' _ | ' _ \ / _ ` | \ \ \ \
\ \ / _ _ ) | ( _ ) | | | | | | ( _ | | ) ) ) )
' | _ _ | . _ | _ | _ | _ \ _ , | / / / /
=====| _ | =====| _ | _ / = / _ / _ /
:: Spring Boot ::          (v2.6.2)

2021-12-11 20:17:17.599  INFO 13980 --- [           main]
c.t.a.TamingThymeleafApplication        : Starting
TamingThymeleafApplication using Java 11.0.10 on Wims-MacBook-Pro.local
with PID 13980 (/Users/wdb/Projects/personal/taming-thymeleaf/example-
code/chapter02/01 - Generated project/target/classes started by wdb in
```

```
/Users/wdb/Projects/personal/taming-thymeleaf/example-code/chapter02/01
- Generated project)
2021-12-11 20:17:17.600  INFO 13980 --- [           main]
c.t.a.TamingThymeleafApplication : No active profile set,
falling back to default profiles: default
2021-12-11 20:17:18.062  INFO 13980 --- [           main]
o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with
port(s): 8081 (http)
2021-12-11 20:17:18.069  INFO 13980 --- [           main]
o.apache.catalina.core.StandardService : Starting service [Tomcat]
2021-12-11 20:17:18.069  INFO 13980 --- [           main]
org.apache.catalina.core.StandardEngine : Starting Servlet engine:
[Apache Tomcat/9.0.55]
2021-12-11 20:17:18.108  INFO 13980 --- [           main]
o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded
WebApplicationContext
2021-12-11 20:17:18.108  INFO 13980 --- [           main]
w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext:
initialization completed in 480 ms
2021-12-11 20:17:18.246  WARN 13980 --- [           main]
ion$DefaultTemplateResolverConfiguration : Cannot find template
location: classpath:/templates/ (please add some templates or check your
Thymeleaf configuration)
2021-12-11 20:17:18.298  INFO 13980 --- [           main]
o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s):
8081 (http) with context path ''
2021-12-11 20:17:18.306  INFO 13980 --- [           main]
c.t.a.TamingThymeleafApplication : Started
TamingThymeleafApplication in 0.944 seconds (JVM running for 1.173)
```

What we see is the start of an embedded Tomcat running on port 8080. How much code does this need? Very little if we check out the [TamingThymeleafApplication.java](#) file:

```
package com.tamingthymeleaf.application;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class TamingThymeleafApplication {

    public static void main(String[] args) {
        SpringApplication.run(TamingThymeleafApplication.class, args);
    }
}
```

}

The `@SpringBootApplication` marks the class as a Spring Boot application and that will trigger all the magic of Spring Boot at startup. It scans the classpath to see what dependencies are available and will configure everything according to sensible defaults.

One of those defaults is the port 8080. Luckily, everything in Spring Boot is [configurable in a multitude of ways](#).

As an example, you can change the port by adding the following to the `application.properties` file:

`application.properties`

`server.port=8081`

Restart the application and Tomcat will now run at 8081 instead.



Two other important ways to configure properties are using the command line or via environment variables. You can view a list of the most common properties at [Appendix A. Common application properties](#) of the Spring Boot reference documentation.

There is still a lot more to learn about building and deploying Spring Boot applications, but you will learn more about them as needed as we create our application throughout the book.

## 2.3. Summary

In this chapter, you learned:

- How to create a Spring Boot application using the Spring Initializer.
- How to configure the Tomcat port of the embedded container.



If you ever get stuck following along, you can refer to the full source code on GitHub:  
<https://github.com/wimdeblauwe/taming-thymeleaf-sources>

[1] Selecting *Jar* will produce a standalone application with an embedded web server. Select *War* if you need to deploy to a standalone web server.

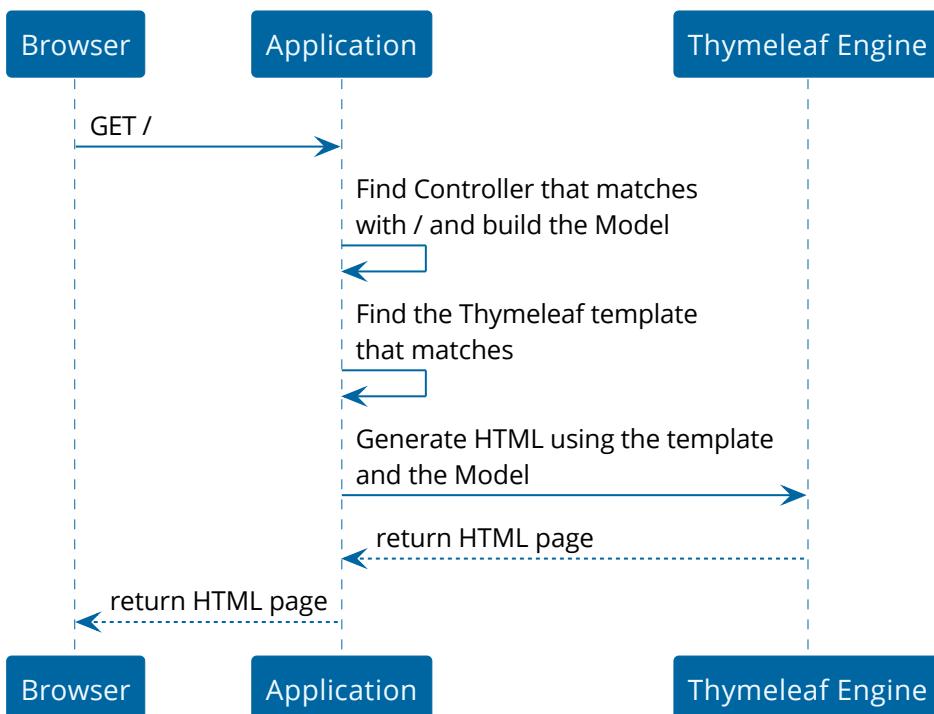
# Chapter 3. Thymeleaf introduction

## 3.1. 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.<sup>[1]</sup>

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.

## 3.2. 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:

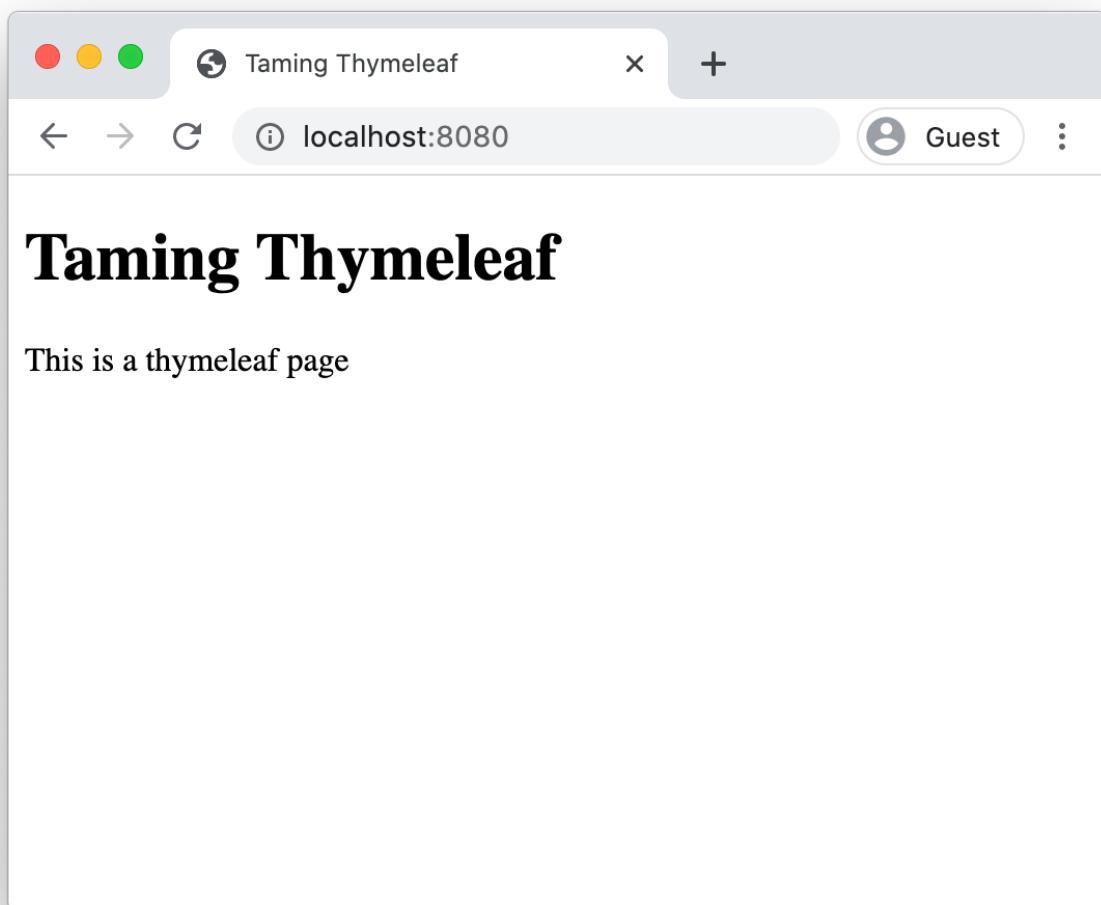


Figure 2. Thymeleaf rendering plain HTML page

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:

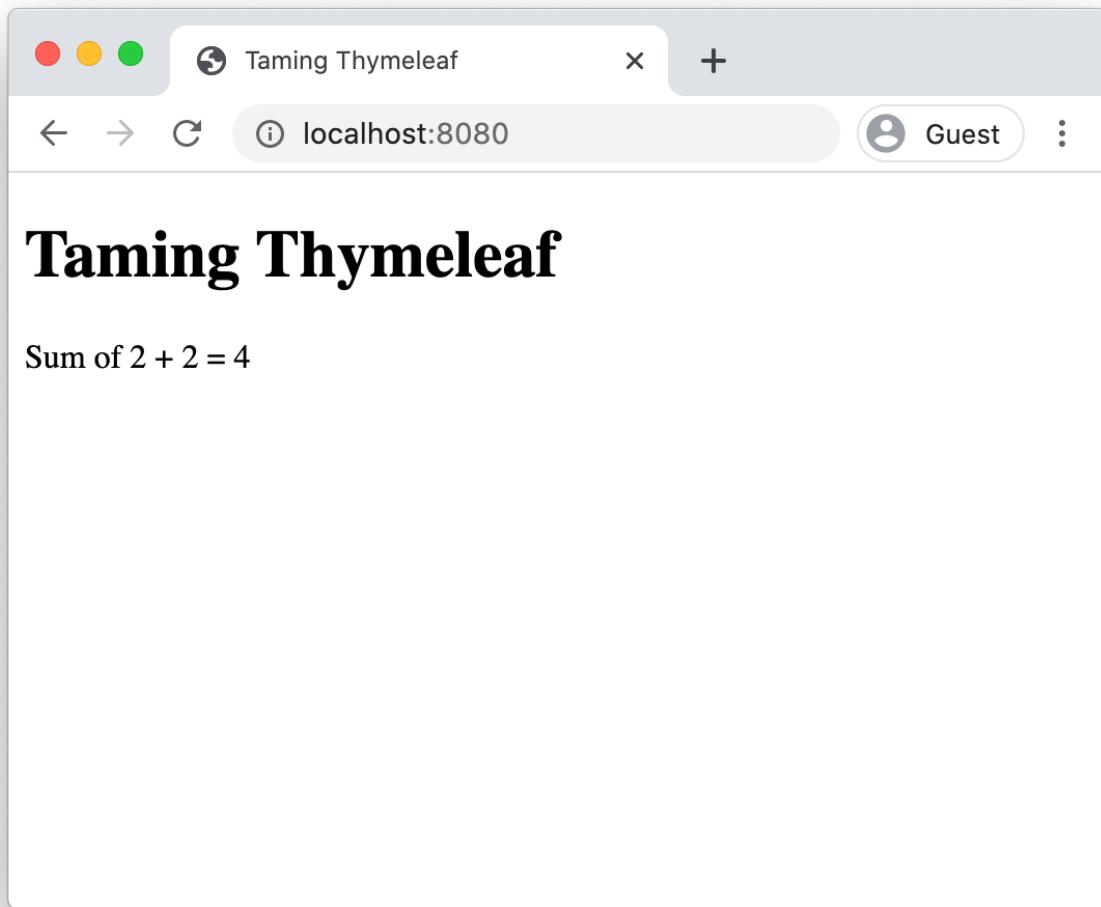


Figure 3. Thymeleaf rendering summation expression

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.

## 3.3. 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:

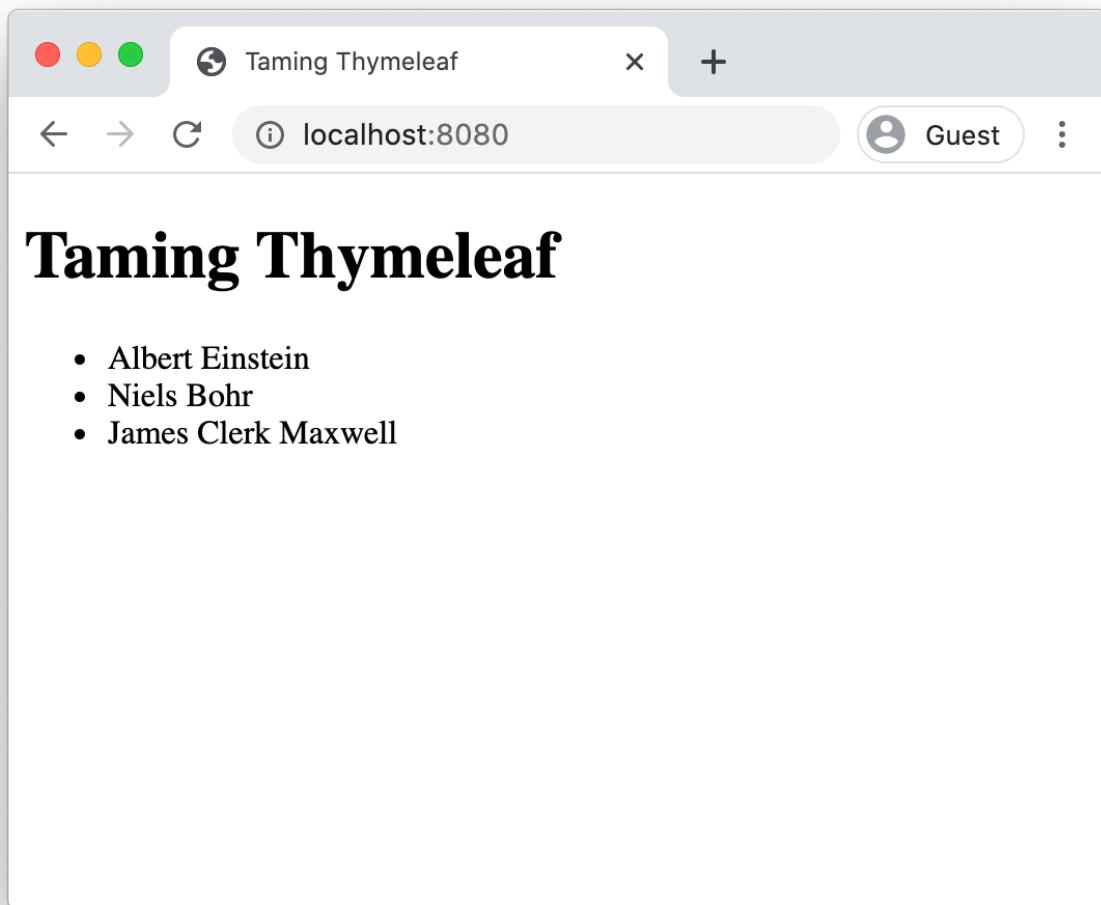


Figure 4. Using a controller to supply data to the view

## 3.4. 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. Luckily, for most cases, the syntax is exactly the same.

### 3.4.1. 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>
```

## 3.4.2. 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.

### 3.4.3. 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.

### 3.4.4. 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>
```

### 3.4.5. 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>
```

### 3.4.6. 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>
```

## 3.5. 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.

### 3.5.1. 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**.

### 3.5.2. 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**.

### 3.5.3. 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.

### 3.5.4. 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 2<sup>nd</sup> one will be rendered depending on the value of `user.followerCount`.

### 3.5.5. 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.

## 3.6. 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>
```

## 3.7. 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.

# Chapter 4. Thyme Wizards

In this chapter, we are going to start the real work of our application. The application is a made-up CMS (Content Management System) that the *Thyme Wizards*, a local basketball team will use to track players, coaches, games, ...



In this chapter and all following ones, the book will always explain the general concepts first and then apply them to our example application. Keep that in mind, so you don't immediately try to apply the concepts. The book will guide you step-by-step along the path to Thymeleaf mastery.

## 4.1. Cascading Style Sheets

To style a web application, you need to use Cascading Style Sheets, or CSS in short. The easiest way to add CSS to a Spring Boot application is adding your CSS file in the `src/main/resources/static` directory. Anything that is present in that directory will be served through the embedded Tomcat container.

To try this out, add the following CSS file:

`src/main/resources/static/css/application.css`

```

h1 {
    color: #5f5f5f;
    border-bottom: 5px solid darkseagreen;
}

ul {
    font-variant: small-caps;
}

```

To have our HTML use the CSS, we need to link to it in the template:

`src/main/resources/templates/index.html`

```

<!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>
    <link rel="stylesheet" th:href="@{/css/application.css}"/> ①
</head>

...

```

- ① Link to the `/css/application.css` file. We must use a path relative to the `src/main/resources/static` directory here.

The result:

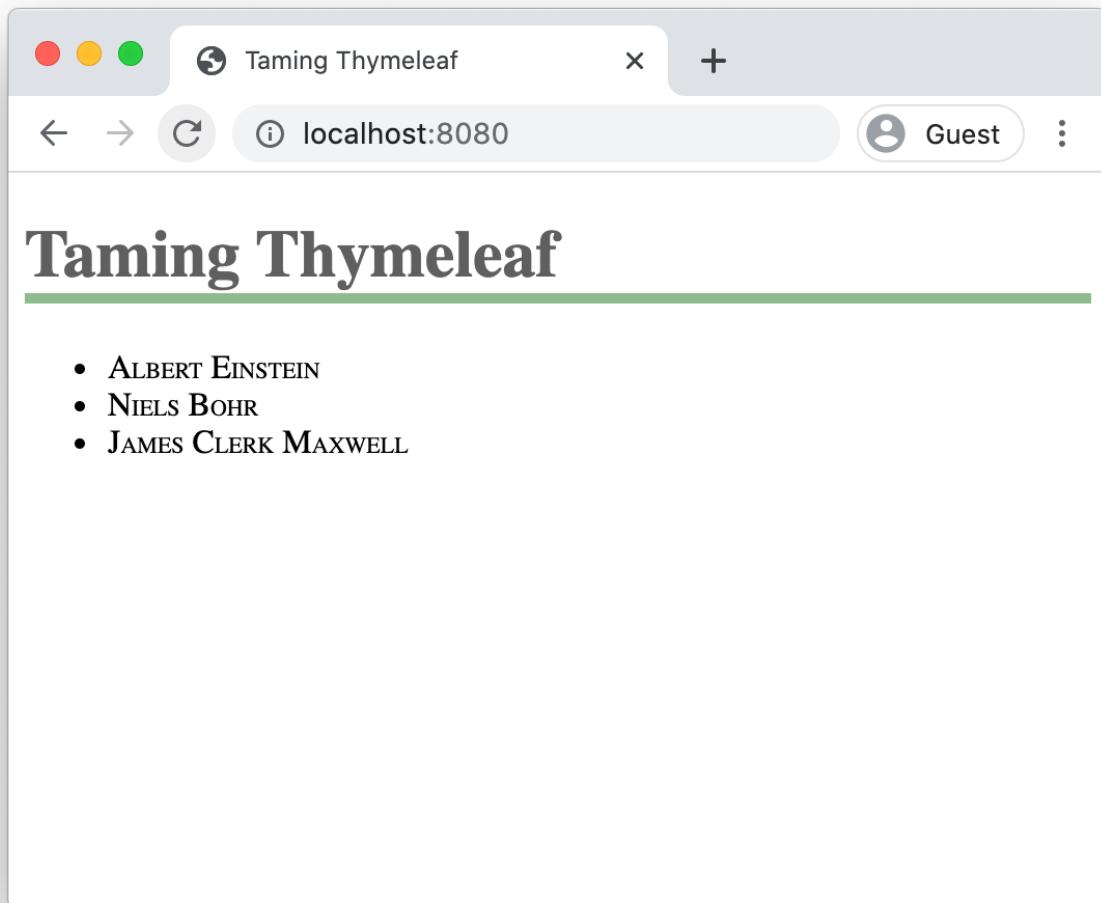


Figure 5. CSS example

## 4.2. Tailwind CSS

### 4.2.1. What is Tailwind CSS?

For the look of the application, we will use [Tailwind CSS](#) and the [Tailwind UI](#) components.

Tailwind CSS is a utility-first CSS framework. You will learn about it as we build the application, but if you want some good introduction, checkout [the very informative screencasts](#) on the website.

In a nutshell, the goal of Tailwind is that you need almost no custom CSS. You apply ready-made classes to your HTML.

Some examples of Tailwind CSS classes:

- `ml-6` → `margin-left: 1.5rem`

- `flex` → `display: flex`
- `bg-white` → `background-color: white`

It looks something like this when applied in HTML:

```
<div class="p-6 max-w-sm mx-auto bg-white rounded-xl shadow-md flex items-center space-x-4">
  <div class="flex-shrink-0">
    
  </div>
  <div>
    <div class="text-xl font-medium text-black">ChitChat</div>
    <p class="text-gray-500">You have a new message!</p>
  </div>
</div>
```

Which renders to:

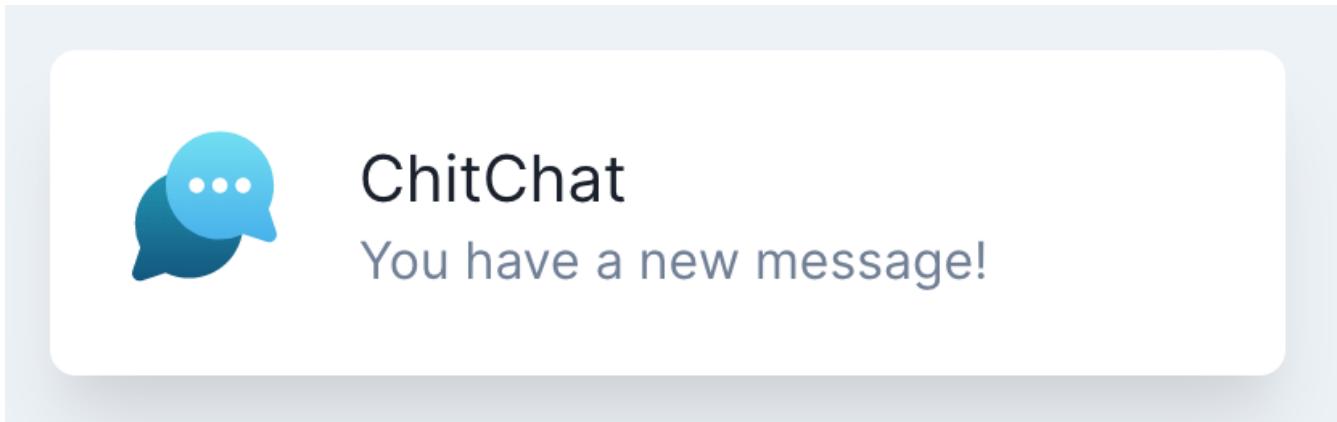


Figure 6. Tailwind example

As stated on the Tailwind website itself as well: You are probably thinking that this is a bad idea to have all those class in the HTML. I can assure you that after working with it on an actual project, I would not use anything else anymore. See [Utility first](#) for a more detailed explanation why:

*But once you've actually built something this way, you'll quickly notice some really important benefits:*

- **You aren't wasting energy inventing class names.** No more adding silly class names like `sidebar-inner-wrapper` just to be able to style something, and no more agonizing over the perfect abstract name for something that's really just a flex container.
- **Your CSS stops growing.** Using a traditional approach, your CSS files get bigger every time you add a new feature. With utilities, everything is reusable so you rarely need to write new CSS.
- **Making changes feels safer.** CSS is global and you never know what you're breaking when you make a change. Classes in your HTML are local, so you can change them without worrying about something else breaking.

If you are doubting, give it the benefit of the doubt as I did when I started out with it. I am sure it will grow on you.

## 4.2.2. Adding Tailwind CSS

To start using Tailwind CSS, we will need to add npm to our application.

### 4.2.2.1. Npm installation

If you don't have npm installed, do so now by following the instructions at [Downloading and installing Node.js and npm](#)

*macOS or Linux*

If you are on macOS or Linux, install [nvm](#) (Node Version Manager) first:

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.0/install.sh  
| bash
```

Next, use it to install npm:

```
nvm install-latest-npm
```

Check the version to see if npm was installed properly:

```
> npm --version  
8.3.0
```

*Windows*

On Windows, it is best to use [nvm-windows](#) as node version manager.

After installing it, use the command line to install npm:

```
nvm install latest
```

Check the version to see if npm was installed properly:

```
> npm --version  
8.3.0
```

### 4.2.2.2. Initialize npm

In the root of the project, create a [package.json](#) file:

*package.json*

```
{
```

```
"name": "taming-thymeleaf-app"
}
```

We can now install Tailwind as a dependency:

```
npm install -D tailwindcss@latest postcss@latest autoprefixer@latest
```

This command will do 3 things:

1. Update `package.json`:

```
{
  "name": "taming-thymeleaf-app",
  "devDependencies": {
    "autoprefixer": "^10.4.0",
    "postcss": "^8.4.4",
    "tailwindcss": "^3.0.1"
  }
}
```

2. Create a `package-lock.json` file. This file should be committed as it contains the *exact* version of each dependency that the application will use.
3. Download tailwindcss into the `node_modules` directory.

#### 4.2.2.3. Add Tailwind as a PostCSS plugin

Create a `postcss.config.js` file at the root of the project:

```
module.exports = {
  plugins: {
    tailwindcss: {},
    autoprefixer: {}
  }
}
```

This adds `tailwindcss` and `autoprefixer` as PostCSS plugins.

#### 4.2.2.4. Update the CSS file

We now need to update our `application.css` file to use the tailwind classes:

```
@tailwind base;
@tailwind components;
```

```
@tailwind utilities;
```

Our build system will now have to turn those `@tailwind` directives into actual CSS that the browser will understand.

To make that possible, we will use [Gulp](#).

Install Gulp with:

```
npm install --global gulp-cli
```

Check the version:

```
> gulp --version
CLI version: 2.3.0
```

Now install gulp (along with some other dependencies we will need) as a development dependency:

```
npm install --save-dev gulp gulp-watch browser-sync gulp-babel \
@babel/core @babel/preset-env \
gulp-terser gulp-uglifycss gulp-postcss gulp-purgecss gulp-environments
```

Next, create a [gulpfile.js](#) at the root of the project with the following contents:

```
const gulp = require('gulp');
const babel = require("gulp-babel");
const watch = require('gulp-watch');
const browserSync = require('browser-sync').create();
const environments = require('gulp-environments');
const uglifycss = require('gulp-uglifycss');
const terser = require('gulp-terser');
const postcss = require('gulp-postcss');
const purgecss = require('gulp-purgecss');

const production = environments.production;

gulp.task('watch', () => {
  browserSync.init({
    proxy: 'localhost:8080',
  });

  gulp.watch(['src/main/resources/**/*.*.html'], gulp.series('copy-
html+css-and-reload'));
  gulp.watch(['src/main/resources/**/*.*.css'], gulp.series('copy-css-
```

```

and-reload'));
    gulp.watch(['src/main/resources/**/*.js'], gulp.series('copy-js-and-
reload'));
});

gulp.task('copy-html', () =>
    gulp.src(['src/main/resources/**/*.html'])
        .pipe(gulp.dest('target/classes/'))
);

gulp.task('copy-css', () =>
    gulp.src(['src/main/resources/**/*.css'])
        .pipe(postcss())
        .pipe(production(uglifycss()))
        .pipe(gulp.dest('target/classes/'))
);

gulp.task('copy-js', () =>
    gulp.src(['src/main/resources/**/*.js'])
        .pipe(babel())
        .pipe(production(terser()))
        .pipe(gulp.dest('target/classes/'))
);

// When the HTML changes, we need to copy the CSS also because
// the Tailwind CSS JIT compiler might generate new CSS
gulp.task('copy-html+css-and-reload', gulp.series('copy-html', 'copy-
css', reload));
gulp.task('copy-css-and-reload', gulp.series('copy-css', reload));
gulp.task('copy-js-and-reload', gulp.series('copy-js', reload));

gulp.task('build', gulp.series('copy-html', 'copy-css', 'copy-js'));
gulp.task('default', gulp.series('watch'));

function reload(done) {
    browserSync.reload();
    done();
}

```

This configures Gulp to have 2 main tasks:

- **build**: builds the HTML, Javascript and CSS and copies it to the `target/classes` directory where Spring Boot expects them
- **watch**: Watches the HTML, Javascript and CSS source files for changes and automatically runs `build` when they changed.

To start the Gulp tasks via npm, we add the following scripts to `package.json`:

```
{
  "name": "taming-thymeleaf-app",
  "scripts": {
    "watch": "gulp watch",
    "build": "gulp build",
    "build-prod": "NODE_ENV='production' gulp build --env production"
  },
  ...
}
```

One last thing before we can run the scripts is the configuration of Tailwind.

Tailwind uses a Just In Time (JIT) compiler that will only generate the classes that are actually used in the HTML. To make this work, we need to configure Tailwind so it knows where our HTML files are located.

Generate a default `tailwind.config.js` config file by running:

```
npx tailwind init
```



The `npx` command is bundled with NPM. It allows to execute tools from the npm registry without having to install them. See <https://blog.npmjs.org/post/162869356040/introducing-npx-an-npm-package-runner> for more detailed information.

Now update the file so that the purging is aware of the Thymeleaf templates:

`tailwind.config.js`

```
module.exports = {
  content: ['./src/main/resources/templates/**/*.{html}'],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

We also have `uglifycss` configured. This will compress the CSS as much as possible by removing whitespace.

Now run `npm run build` to run the build. If all goes well, there should be an `target/classes/static/css/application.css` file present that contains the generated CSS.

To build for production, run:

```
npm run build-prod
```

The resulting `application.css` is only 1 line now:

```
*,:before,:after{box-sizing:border-box;border-width:0;border-style:  
:solid;border-color:currentColor}...
```

#### 4.2.3. Configure Maven

We are not fully done yet, because if you would try to start the application now from the IDE, the `application.css` in `target/classes/static/css` will look exactly the source we wrote without any of the processing. Let us configure Maven so everything works nicely together.

Start with adding the `frontend-maven-plugin` to the `pom.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>  
<project xmlns="http://maven.apache.org/POM/4.0.0"  
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0  
    https://maven.apache.org/xsd/maven-4.0.0.xsd">  
    <modelVersion>4.0.0</modelVersion>  
    <parent>  
        <groupId>org.springframework.boot</groupId>  
        <artifactId>spring-boot-starter-parent</artifactId>  
        <version>2.6.2</version>  
        <relativePath/> <!-- lookup parent from repository -->  
    </parent>  
    <groupId>com.tamingthymeleaf</groupId>  
    <artifactId>taming-thymeleaf-application</artifactId>  
    <version>0.0.1-SNAPSHOT</version>  
    <name>Taming Thymeleaf 04.02</name>  
    <description>Example application for the Taming Thymeleaf  
book</description>  
  
    <properties>  
        <java.version>17</java.version>  
  
        <frontend-maven-plugin.version>1.12.0</frontend-maven-  
plugin.version> ①  
        <frontend-maven-plugin.nodeVersion>v16.13.1</frontend-maven-  
plugin.nodeVersion> ②  
        <frontend-maven-plugin.npmVersion>8.1.2</frontend-maven-  
plugin.npmVersion> ③
```

```

</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-thymeleaf</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>

<build>
    <resources>
        <resource>
            <directory>src/main/resources</directory>
            <excludes> ④
                <exclude>**/*.html</exclude>
                <exclude>**/*.css</exclude>
                <exclude>**/*.js</exclude>
            </excludes>
        </resource>
    </resources>
    <pluginManagement>
        <plugins>
            <plugin>
                <groupId>com.github.eirslett</groupId>
                <artifactId>frontend-maven-plugin</artifactId>
                <version>${frontend-maven-plugin.version}</version>
                <executions>
                    <execution> ⑤
                        <id>install-frontend-tooling</id>
                        <goals>
                            <goal>install-node-and-npm</goal>
                        </goals>
                        <configuration>
                            <nodeVersion>${frontend-maven-
plugin.nodeVersion}</nodeVersion>

```

```
<npmVersion>${frontend-maven-
plugin.npmVersion}</npmVersion>
    </configuration>
</execution>
<execution>⑥
    <id>run-gulp-build</id>
    <goals>
        <goal>npm</goal>
    </goals>
    <configuration>
        <arguments>run build</arguments>
    </configuration>
</execution>
</executions>
</plugin>
</plugins>
</pluginManagement>
<plugins>
    <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
    <plugin> ⑦
        <groupId>com.github.eirslett</groupId>
        <artifactId>frontend-maven-plugin</artifactId>
    </plugin>
</plugins>
</build>
<profiles>
    <profile>
        <id>release</id>
        <build>
            <plugins>
                <plugin>
                    <groupId>com.github.eirslett</groupId>
                    <artifactId>frontend-maven-plugin</artifactId>
                    <executions>
                        <execution> ⑧
                            <id>run-gulp-build</id>
                            <goals>
                                <goal>npm</goal>
                            </goals>
                            <configuration>
                                <arguments>run build-
prod</arguments>
```

```

        </configuration>
    </execution>
</executions>
</plugin>
</plugins>
</build>
</profile>
</profiles>
</project>

```

- ① Specify the `frontend-maven-plugin` version to use in a Maven property.
- ② Specify the version of node to use (Use `node --version` to find out your current installed version).
- ③ Specify the version of npm to use (Use `npm --version` to find out your current installed version).
- ④ Configure excludes for all HTML, Javascript and CSS files. We will copy those to the `target/classes` output directory via the `frontend-maven-plugin`, so we don't want Maven itself to copy those.
- ⑤ Configure the first execution of the `frontend-maven-plugin` to install `npm` and `node` so the build also works without having those tools installed.
- ⑥ Configure the second execution of the `frontend-maven-plugin` to run the Gulp build
- ⑦ Add the `frontend-maven-plugin` to run by default (It is bound to the `generate-resources` Maven lifecycle phase by default).
- ⑧ Configure Maven so that when using the `release` profile, the `npm run build-prod` is run so we have our minified CSS when building a release version.

To test, run `mvn verify` and check the contents of `target/classes/static/css/application.css`.

For comparison, run `mvn verify -Prelease` after that and check the contents of the CSS file again. The file should be minified.

#### 4.2.4. Live reload

With this setup, we also now have support for live reload of the browser when HTML, Javascript or CSS changes. To make that fully work, we need to disable the caching of Thymeleaf templates that is enabled by default. Additionally, we also implement a cache busting mechanism using the `spring.web.resources.chain.strategy.content.*` properties.

Add a new `application-local.properties` file:

`src/main/resources/application-local.properties`

```

spring.thymeleaf.cache=false
spring.web.resources.chain.strategy.content.enabled=true
spring.web.resources.chain.strategy.content.paths=/**
```

Now start the Spring Boot application with the `Local` profile enabled. If you use IntelliJ, you can do

this from the run configuration dialog, at the *Active profiles* setting:

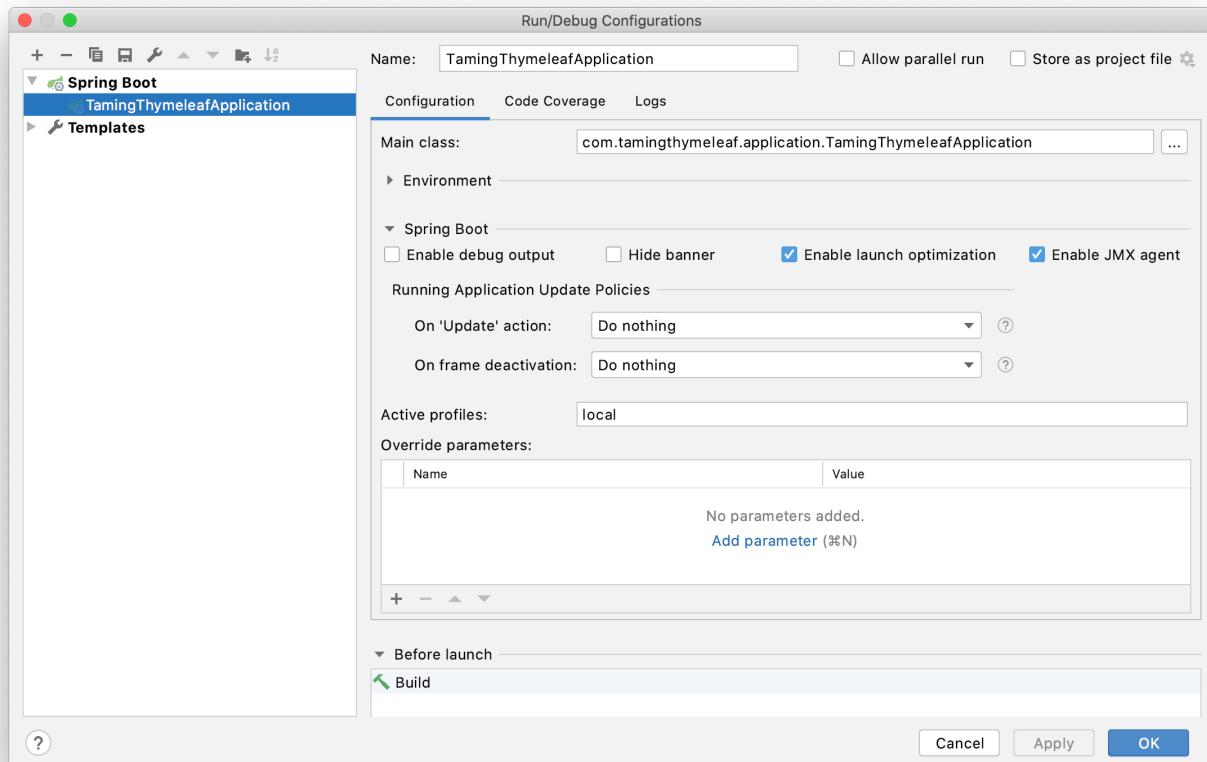


Figure 7. IntelliJ IDEA Run Configuration dialog

Finally, open a command line terminal at the root of the project and run:

```
npm run build && npm run watch
```

This will first build the frontend and then watch for any changes. It will also automatically open your default browser at <http://localhost:3000>

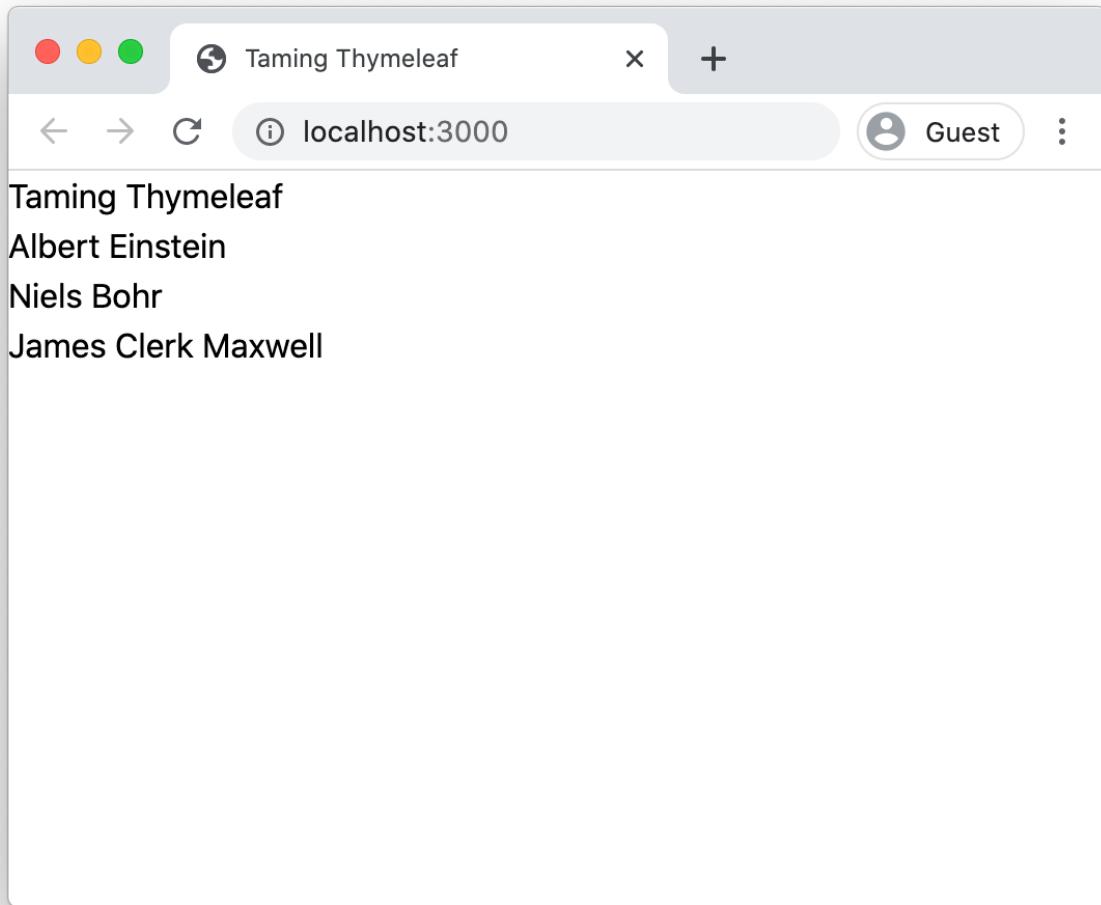


Figure 8. Live reload in browser with default Tailwind CSS styling

If you look at the Developer Tools of your browser, you will see that not `application.css` is loaded, but something like `application-20a16208bdedf3ce24834bc96a8374d4.css`.

Thymeleaf will also have updated the link to the CSS correspondingly:

```
<link rel="stylesheet" href="/css/application-
20a16208bdedf3ce24834bc96a8374d4.css">
```

By having a unique name each time the CSS changes, we ensure the live reload works fine.

To test out the live reload, edit the `index.html`. For example, change the `<h1>` tag to be:

```
<h1 th:text="${pageTitle}" class="text-4xl mx-4 mb-4 border-b-4 border-
green-700">Taming Thymeleaf</h1>
```

Save the file and watch the browser automatically display the updated content. Result after adding some more Tailwind CSS classes:

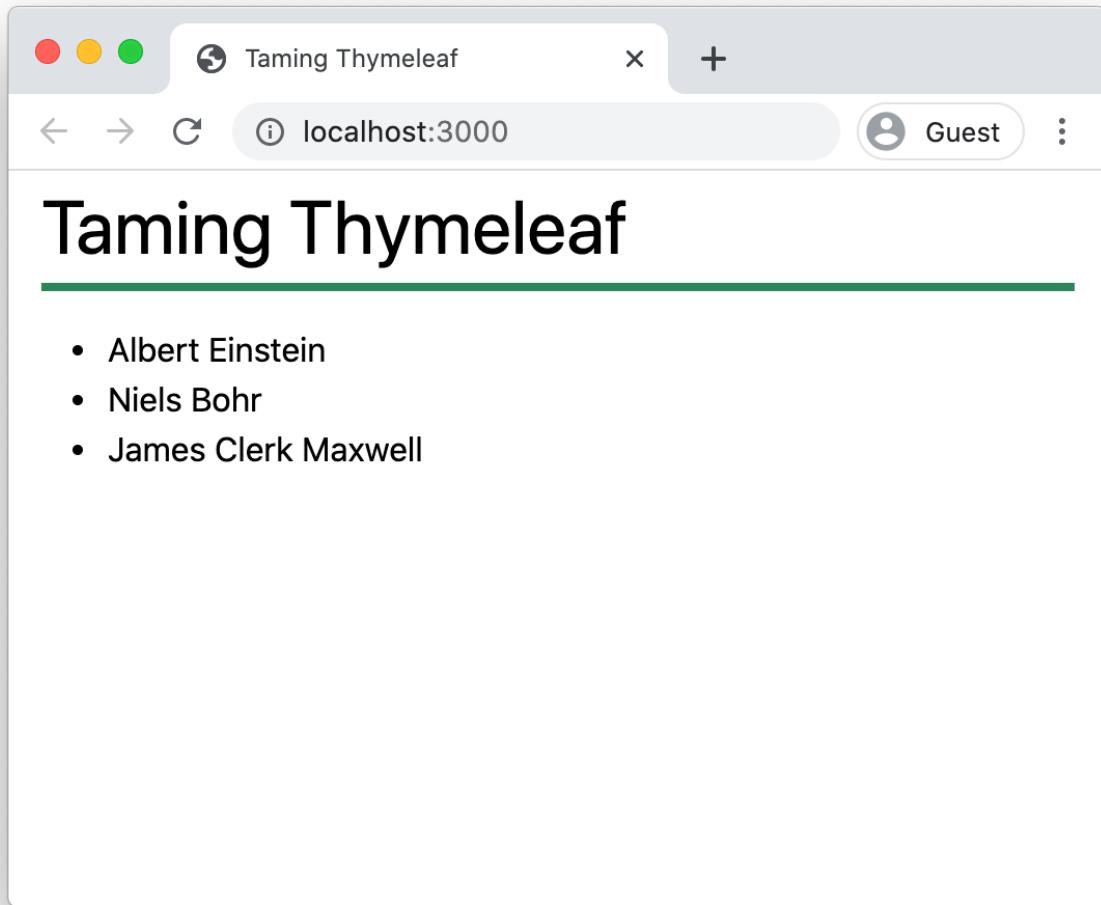


Figure 9. Live reload in browser with some Tailwind CSS classes applied

#### 4.2.5. Tailwind CSS design system configuration

The observant reader might have noticed that the green color used to draw the bottom border is *not* the exact same green as when we supplied our own custom CSS. This is because `darkseagreen` is not part of the default *design system* configuration of Tailwind CSS.

This is one of the big advantages of Tailwind CSS. It can be configured to only expose colors, margins, ... that we want to allow according to our own application style.

Let us customize the default Tailwind CSS theme and add our custom color:

```
module.exports = {
  content: ['./src/main/resources/templates/**/*.{html}'],
  theme: {
    extend: {
      colors: {
        'taming-thymeleaf-green': 'darkseagreen'
      }
    }
}
```

```
  },
},
plugins: [],
}
```

After this, we can start using `taming-thymeleaf-green` as a color in our HTML. The JIT compiler from Tailwind CSS will start generating the corresponding classes as we start using them in our HTML.

Use it like this:

```
<h1 th:text="${pageTitle}" class="text-4xl mx-4 mb-4 border-b-4 border-taming-thymeleaf-green">Taming Thymeleaf</h1>
```

Run `npm run build` to generate the CSS again. It should now contain something like this:

```
.border-taming-thymeleaf-green {
  --tw-border-opacity: 1;
  border-color: rgb(143 188 143 / var(--tw-border-opacity));
}
```

And see a result like this in the browser:

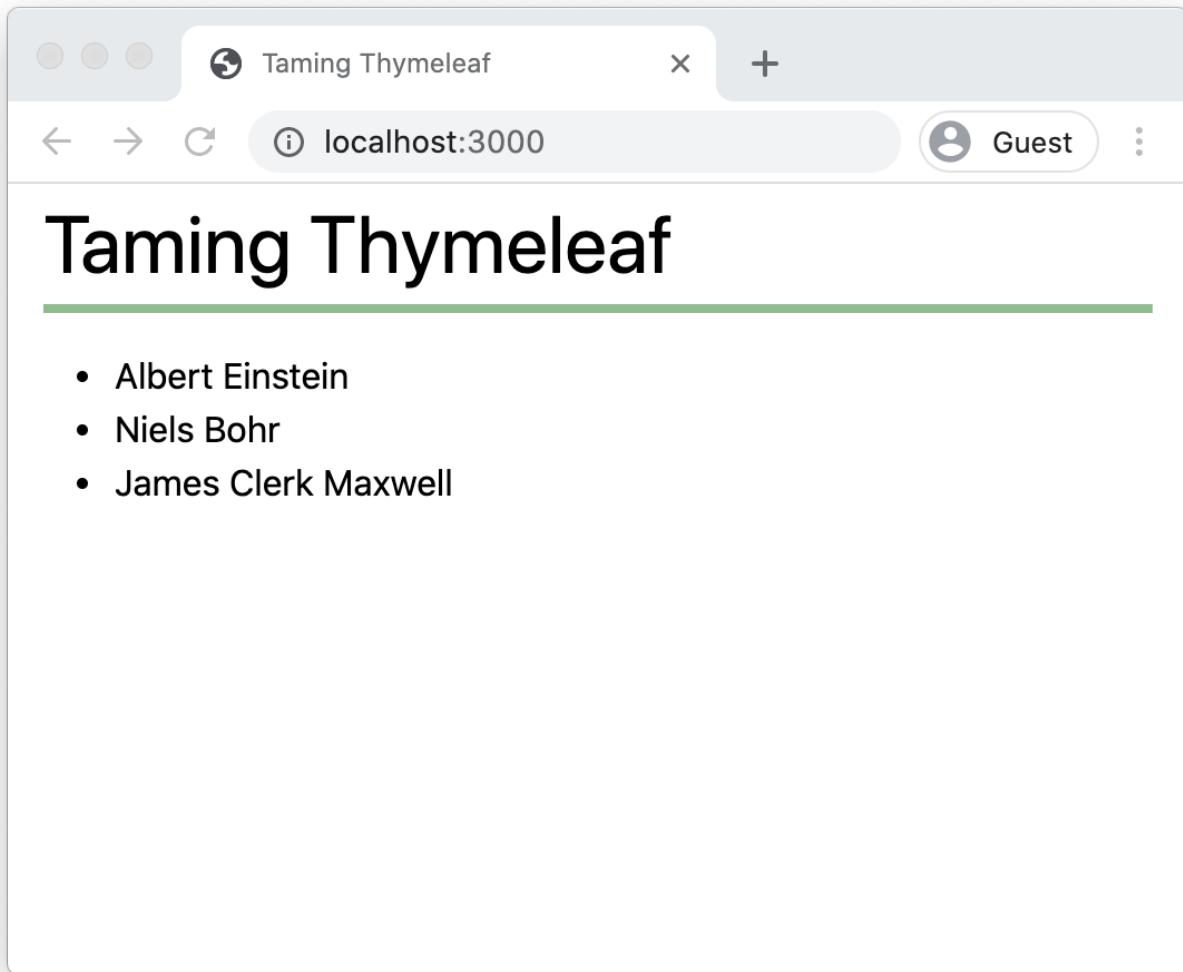


Figure 10. Using Tailwind custom configuration

You can extend the base theme from Tailwind, or create your own custom one from scratch. See [Tailwind CSS configuration](#) for more details on all the ways you can customize Tailwind for your project.

## 4.3. Application shell

### 4.3.1. Tailwind UI

We'll start our application UI by implementing the application shell. This includes the menu items to navigate to each section of the app, and a user avatar with user name to display the currently logged on user.

As this is a fictional application, there is no design from an actual designer to work with. So we will use the designs from [Tailwind UI](#) as this allows us to create something beautiful quickly.

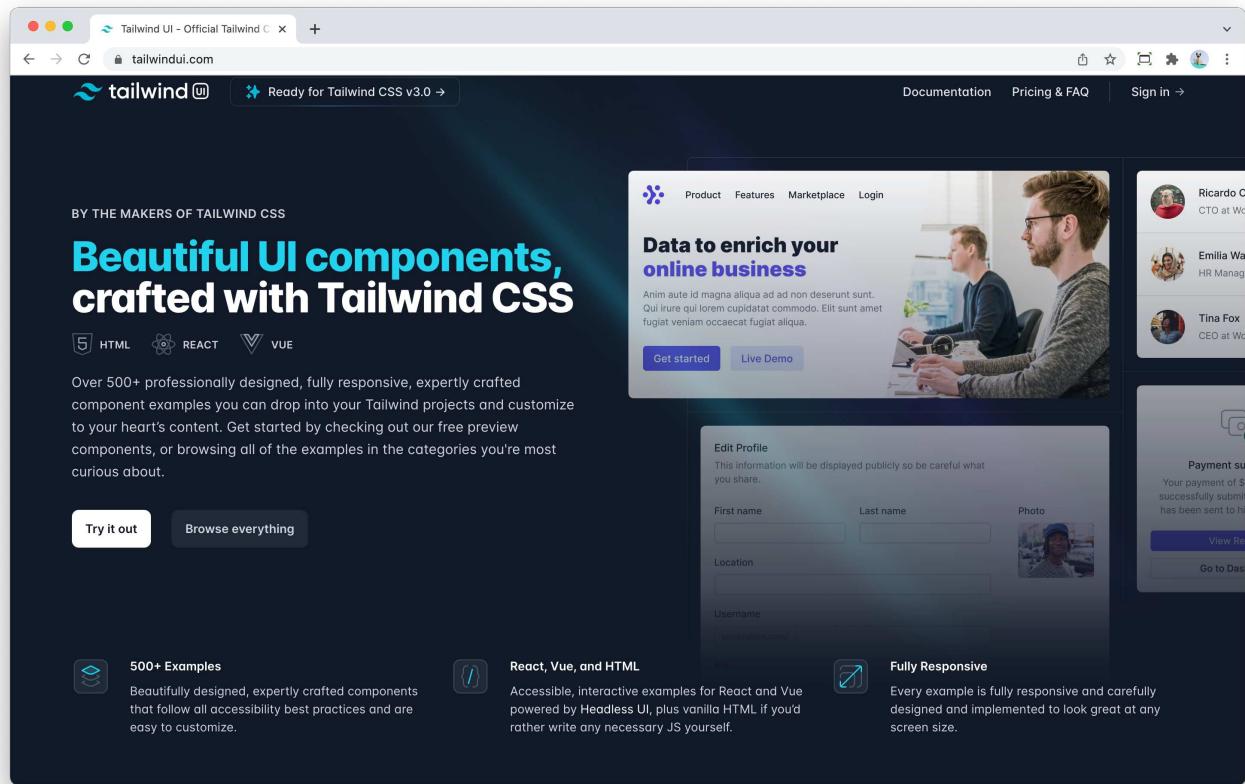


Figure 11. Tailwind UI homepage

Update `tailwind.config.js` to set the Inter font family:

```
const defaultTheme = require('tailwindcss/defaultTheme');

module.exports = {
  content: ['./src/main/resources/templates/**/*.{html}'],
  theme: {
    extend: {
      fontFamily: {
        sans: ['Inter var', ...defaultTheme.fontFamily.sans],
      },
    },
    plugins: [],
  }
};
```

If you have a Tailwind UI license, you can copy the HTML from the Tailwind UI website into the `src/main/resources/templates/index.html` page. I took the *Light sidebar with light header* application shell, but you can use whatever you want obviously. If you don't have a license, you can see the HTML at the end of this section, or copy it from the accompanying sources of the book.

The *Light sidebar with light header* requires the form plugin, so we need to add it to our `package.json` like this:

```
npm install -D @tailwindcss/forms@latest
```

We also update the [tailwind.config.js](#) file to use the plugin:

```
...
plugins: [
    require('@tailwindcss/forms')
],
```

Now run [npm run build](#) and start the application.

Opening the browser at <http://localhost:8080> should show something like this:

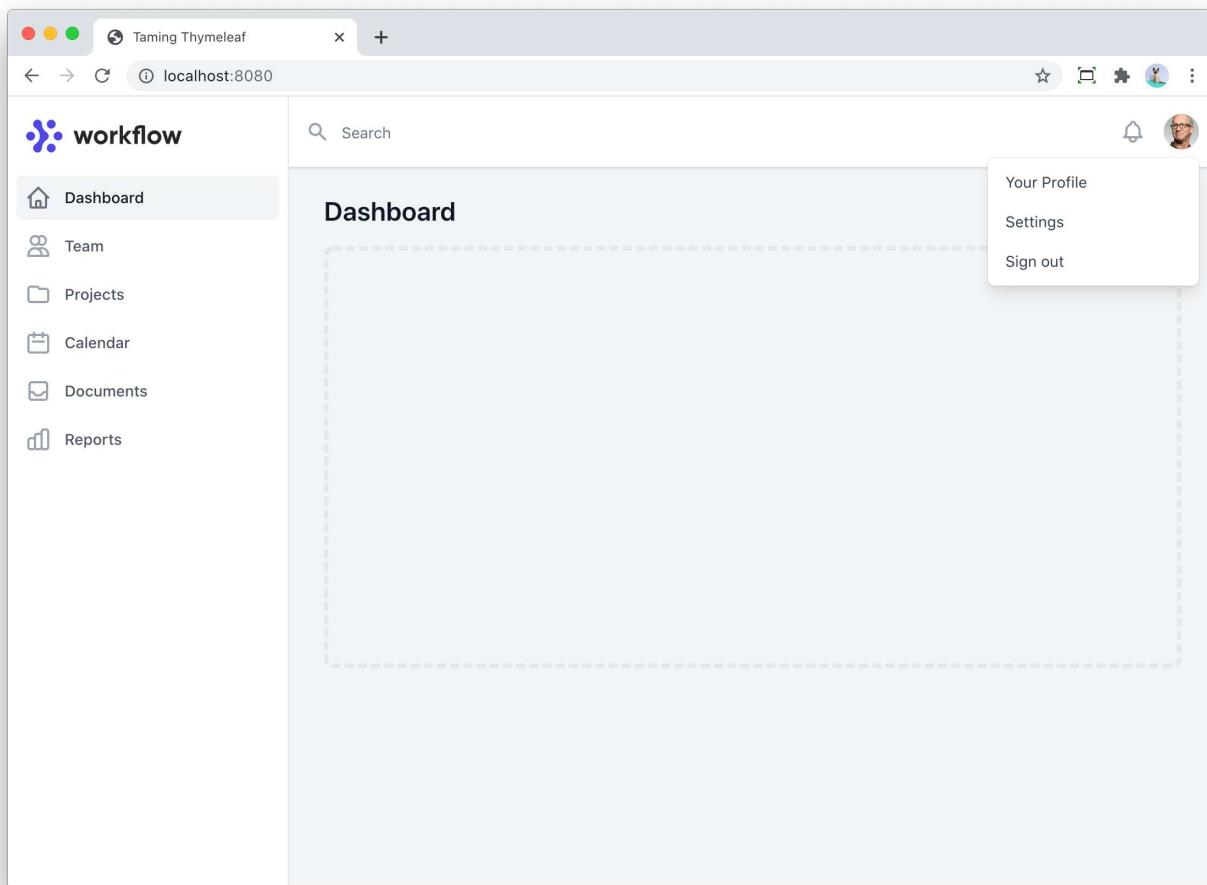


Figure 12. Initial application shell

We can see that the popup menu is shown immediately on page load and cannot be hidden by clicking the avatar.

Let's fix that.

### 4.3.2. Client side interactivity

We will need some minimal Javascript in our application to make the popup menu work. There are a lot of Javascript libraries, but we need something that is light and does not require a full framework setup.

Tailwind UI recommends [AlpineJS](#) for server-rendered website like the one we are building, so we will go with that.

We will also remove the search bar and the notification bell icon from the application shell we choose from Tailwind UI as we won't be using that for now.

To start, we add AlpineJS via a CDN, and add some custom Javascript to control the user popup menu:

```
<script src="https://unpkg.com/alpinejs@3.7.0/dist/cdn.min.js"
defer></script>
<script>
    function userPopupMenu() { ①
        return {
            show: false, ②
            toggleVisibility() { ③
                this.show = !this.show;
            },
            close() { ④
                this.show = false;
            },
            isVisible() { ⑤
                return this.show === true;
            }
        };
    }
</script>
```

① Define the `userPopupMenu()` JavaScript function

② Keep track of the visibility of the popup menu in the `show` variable

③ Defines a method that will allow to toggle the visibility from visible to invisible and back

④ The `close()` method will make the popup invisible

⑤ Returns if the popup menu should be visible or not. This method will be bound via AlpineJS to the HTML element to show and hide.

The relevant HTML part that uses this:

```
<!-- Profile dropdown -->
<div class="ml-3 relative"
    x-data="userPopupMenu()" ①
    @click.away="close" ②
    @keydown.window.escape="close"> ③
```

```

<div>
    <button class="max-w-xs bg-white flex items-center text-sm rounded-full focus:outline-none focus:ring-2 focus:ring-offset-2 focus:ring-indigo-500" id="user-menu" aria-haspopup="true"
        @click="toggleVisibility"④
        <span class="sr-only">Open user menu</span>
        
    </button>
</div>
<!--
    Profile dropdown panel, show/hide based on dropdown state.
-->
<div class="origin-top-right absolute right-0 mt-2 w-48 rounded-md shadow-lg"
    x-show="isVisible()"⑤
    x-cloak>⑥
    ...
</div>
</div>

```

- ① **x-data** declares a new component scope. By returning the result of the `userPopupMenu()` function, we will be able to access the data and methods on all HTML elements inside this `<div>`.
  - ② **@click** is a shortcut for `x-on:click`. With the `.away` modifier, our event handler will be executed when the event originates from a source different than itself (or its children). So `@click.away="close"` ensures the popup is closed when the user clicks anywhere else in the application.
  - ③ **@keydown.window.escape** also binds the `close` function as an event listener when somebody presses the ESC key.
  - ④ **@click** allows to configure an event listener when the `<button>` is clicked. In this case, it will toggle the visibility each time the button is clicked.
  - ⑤ **x-show** allows to bind an expression to the `display: none;` style for the element. By binding to `isVisible`, we will effectively show and hide the menu component.
  - ⑥ **x-cloak**: When JavaScript is disabled, the popup menu should not be shown by default. By using `x-cloak` combined with a small bit of custom CSS, we avoid that the popup menu is visible. See `x-cloak` on the Alpine.js website for more info.
- Be sure to update `application.css` as well:

```

@tailwind base;
@tailwind components;
@tailwind utilities;

```

```
@layer utilities {
    [x-cloak] {
        display: none;
    }
}
```

With this minimal JavaScript, we implemented the following behaviours for our popup user menu:

- The popup is not visible by default.
- The popup appears when clicking on the avatar.
- The popup disappears again when visible when clicking on the avatar.
- The popup disappears when visible when clicking anywhere else in the application.
- The popup closes when pressing the ESC key.

To make the opening and closing a bit fancier, we can add a bit of transition. Using AlpineJS, this is as simple as adding `.transition` to the `x-show` directive:

```
<div class="origin-top-right absolute right-0 mt-2 w-48 rounded-md shadow-lg"
    x-show.transition="isVisible()"
    x-cloak>
```

This uses a 150ms fade-in and a scale from 95% to 100%.

If we want to fully follow the Tailwind UI recommended transitions, we need to do a bit more work. See the [Alpine specific Tailwind UI documentation](#) for more details.

For this case, we would need to do something like:



```
<div class="origin-top-right absolute right-0 mt-2 w-48 rounded-md shadow-lg"
    x-show="isVisible()"
    x-cloak
    x-transition:enter="transition ease-out duration-100 transform"
        x-transition:enter-start="opacity-0 scale-95"
        x-transition:enter-end="opacity-100 scale-100"
        x-transition:leave="transition ease-in duration-75 transform"
            x-transition:leave-start="opacity-100 scale-100"
            x-transition:leave-end="opacity-0 scale-95"
    >
```

We won't go into detail for the left menu, but this is the resulting `index.html` after implementing the

left side menu for mobile and the top user menu:

```
<!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>
    <meta http-equiv="X-UA-Compatible" content="IE=edge"/>
    <meta name="viewport" content="width=device-width, initial-
scale=1"/>

    <link rel="stylesheet" href="https://rsms.me/inter/inter.css">
    <link rel="stylesheet" th:href="@{/css/application.css}">
</head>
<body>
<div class="h-screen flex overflow-hidden bg-gray-100"
     x-data="sidebarMenu()"
     @keydown.window.escape="closeSidebar">
    <!-- Off-canvas menu for mobile -->
    <div class="md:hidden"
        x-show="isVisible()">
        <div class="fixed inset-0 flex z-40">
            <!--
                Off-canvas menu overlay
            -->
            <div class="fixed inset-0" aria-hidden="true"
                x-show="isVisible()"
                x-cloak
                x-transition:enter="transition-opacity ease-linear
duration-300"
                x-transition:enter-start="opacity-0"
                x-transition:enter-end="opacity-100"
                x-transition:leave="transition-opacity ease-linear
duration-300"
                x-transition:leave-start="opacity-100"
                x-transition:leave-end="opacity-0">
                <div class="absolute inset-0 bg-gray-600 opacity-
75"></div>
            </div>
            <!--
                Off-canvas menu
            -->
            <div class="relative flex-1 flex flex-col max-w-xs w-full
h-full">
        </div>
    </div>
</div>
```

```

pt-5 pb-4 bg-white"
    x-show="isVisible()"
    x-cloak
    x-transition:enter="transition ease-in-out duration-300
transform"
    x-transition:enter-start="-translate-x-full"
    x-transition:enter-end="translate-x-0"
    x-transition:leave="transition ease-in-out duration-300
transform"
    x-transition:leave-start="translate-x-0"
    x-transition:leave-end="-translate-x-full">
    <div class="absolute top-0 right-0 -mr-12 pt-2">
        <!--
            Close sidebar menu button
        -->
        <button class="ml-1 flex items-center justify-center
h-10 w-10 rounded-full focus:outline-none focus:ring-2 focus:ring-inset
focus:ring-white">
            @click="closeSidebar">
            <span class="sr-only">Close sidebar</span>
            <!-- Heroicon name: x -->
            <svg class="h-6 w-6 text-white"
                xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0 24 24"
                stroke="currentColor" aria-hidden="true">
                <path stroke-linecap="round" stroke-
                linejoin="round" stroke-width="2" d="M6 18L18 6M6 6l12 12" />
            </svg>
        </button>
    </div>
    <div class="flex-shrink-0 flex items-center px-4">
        
    </div>
    <div class="mt-5 flex-1 h-0 overflow-y-auto">
        <nav class="px-2 space-y-1">
            <!-- Current: "bg-gray-100 text-gray-900",
Default: "text-gray-600 hover:bg-gray-50 hover:text-gray-900" -->
            <a href="#" class="bg-gray-100 text-gray-900
group flex items-center px-2 py-2 text-base font-medium rounded-md">
                <!-- Current: "text-gray-500", Default:
                "text-gray-400 group-hover:text-gray-500" -->
                <!-- Heroicon name: home -->
                <svg class="text-gray-500 mr-4 h-6 w-6"
                    xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0 24 24"

```

```

stroke="currentColor" aria-hidden="true">
    <path stroke-linecap="round" stroke-
linejoin="round" stroke-width="2" d="M3 12l2-2m0 0l7-7 7 7M5 10v10a1 1 0
001 1h3m10-11l2 2m-2-2v10a1 1 0 01-1 1h-3m-6 0a1 1 0 001-1v-4a1 1 0 011-
1h2a1 1 0 011 1v4a1 1 0 001 1m-6 0h6" />
</svg>
Dashboard
</a>

<a href="#" class="text-gray-600 hover:bg-gray-
50 hover:text-gray-900 group flex items-center px-2 py-2 text-base font-
medium rounded-md">
    <!-- Heroicon name: users -->
    <svg class="text-gray-400 group-hover:text-
gray-500 mr-4 h-6 w-6" xmlns="http://www.w3.org/2000/svg" fill="none"
viewBox="0 0 24 24" stroke="currentColor" aria-hidden="true">
        <path stroke-linecap="round" stroke-
linejoin="round" stroke-width="2" d="M12 4.354a4 4 0 110 5.292M15 21H3v-
1a6 6 0 0112 0v1zm0 0h6v-1a6 6 0 00-9-5.197M13 7a4 4 0 11-8 0 4 4 0 018
0z" />
    </svg>
    Team
</a>

<a href="#" class="text-gray-600 hover:bg-gray-
50 hover:text-gray-900 group flex items-center px-2 py-2 text-base font-
medium rounded-md">
    <!-- Heroicon name: folder -->
    <svg class="text-gray-400 group-hover:text-
gray-500 mr-4 h-6 w-6" xmlns="http://www.w3.org/2000/svg" fill="none"
viewBox="0 0 24 24" stroke="currentColor" aria-hidden="true">
        <path stroke-linecap="round" stroke-
linejoin="round" stroke-width="2" d="M3 7v10a2 2 0 002 2h14a2 2 0 002-
2V9a2 2 0 00-2-2h-6l-2-2H5a2 2 0 00-2 2z" />
    </svg>
    Projects
</a>

<a href="#" class="text-gray-600 hover:bg-gray-
50 hover:text-gray-900 group flex items-center px-2 py-2 text-base font-
medium rounded-md">
    <!-- Heroicon name: calendar -->
    <svg class="text-gray-400 group-hover:text-
gray-500 mr-4 h-6 w-6" xmlns="http://www.w3.org/2000/svg" fill="none"
viewBox="0 0 24 24" stroke="currentColor" aria-hidden="true">

```

```

                <path stroke-linecap="round" stroke-
linejoin="round" stroke-width="2" d="M8 7V3m8 4V3m-9 8h10M5 21h14a2 2 0
002-2V7a2 2 0 00-2-2H5a2 2 0 00-2 2v12a2 2 0 002 2z" />
            </svg>
            Calendar
        </a>

        <a href="#" class="text-gray-600 hover:bg-gray-
50 hover:text-gray-900 group flex items-center px-2 py-2 text-base font-
medium rounded-md">
            <!-- Heroicon name: inbox -->
            <svg class="text-gray-400 group-hover:text-
gray-500 mr-4 h-6 w-6" xmlns="http://www.w3.org/2000/svg" fill="none"
viewBox="0 0 24 24" stroke="currentColor" aria-hidden="true">
                <path stroke-linecap="round" stroke-
linejoin="round" stroke-width="2" d="M20 13V6a2 2 0 00-2-2H6a2 2 0 00-2
2v7m16 0v5a2 2 0 01-2 2H6a2 2 0 01-2-2v-5m16 0h-2.586a1 1 0 00-
.707.293l-2.414 2.414a1 1 0 01-.707.293h-3.172a1 1 0 01-.707-.293l-
2.414-2.414A1 1 0 006.586 13H4" />
            </svg>
            Documents
        </a>

        <a href="#" class="text-gray-600 hover:bg-gray-
50 hover:text-gray-900 group flex items-center px-2 py-2 text-base font-
medium rounded-md">
            <!-- Heroicon name: chart-bar -->
            <svg class="text-gray-400 group-hover:text-
gray-500 mr-4 h-6 w-6" xmlns="http://www.w3.org/2000/svg" fill="none"
viewBox="0 0 24 24" stroke="currentColor" aria-hidden="true">
                <path stroke-linecap="round" stroke-
linejoin="round" stroke-width="2" d="M9 19v-6a2 2 0 00-2-2H5a2 2 0 00-2
2v6a2 2 0 002 2h2a2 2 0 002-2zm0 0V9a2 2 0 012-2h2a2 2 0 012 2v10m-6 0a2
2 0 002 2h2a2 2 0 002-2m0 0V5a2 2 0 012-2h2a2 2 0 012 2v14a2 2 0 01-2
2h-2a2 2 0 01-2-2z" />
            </svg>
            Reports
        </a>
    </nav>
</div>
</div>
<div class="flex-shrink-0 w-14" aria-hidden="true">
    <!-- Dummy element to force sidebar to shrink to fit
close icon -->
</div>

```

```

        </div>
    </div>

    <!-- Static sidebar for desktop -->
    <div class="hidden md:flex md:flex-shrink-0">
        <div class="flex flex-col w-64">
            <div class="flex flex-col flex-grow border-r border-gray-200 pt-5 pb-4 bg-white overflow-y-auto">
                <div class="flex items-center flex-shrink-0 px-4">
                    
                </div>
                <div class="mt-5 flex-grow flex flex-col">
                    <nav class="flex-1 px-2 bg-white space-y-1">
                        <!-- Current: "bg-gray-100 text-gray-900", Default: "text-gray-600 hover:bg-gray-50 hover:text-gray-900" -->
                        <a href="#" class="bg-gray-100 text-gray-900 group flex items-center px-2 py-2 text-sm font-medium rounded-md">
                            <!-- Current: "text-gray-500", Default: "text-gray-400 group-hover:text-gray-500" -->
                            <!-- Heroicon name: home -->
                            <svg class="text-gray-500 mr-3 h-6 w-6" xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0 24 24" stroke="currentColor" aria-hidden="true">
                                <path stroke-linecap="round" stroke-linejoin="round" stroke-width="2" d="M3 12l2-2m0 0l7-7 7 7M5 10v10a1 1 0 001 1h3m10-11l2 2m-2-2v10a1 1 0 00-1 1h-3m-6 0a1 1 0 00-1v-4a1 1 0 00-1h2a1 1 0 001 1v4a1 1 0 001 1m-6 0h6" />
                            </svg>
                            Dashboard
                        </a>

                        <a href="#" class="text-gray-600 hover:bg-gray-50 hover:text-gray-900 group flex items-center px-2 py-2 text-sm font-medium rounded-md">
                            <!-- Heroicon name: users -->
                            <svg class="text-gray-400 group-hover:text-gray-500 mr-3 h-6 w-6" xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0 24 24" stroke="currentColor" aria-hidden="true">
                                <path stroke-linecap="round" stroke-linejoin="round" stroke-width="2" d="M12 4.354a4 4 0 110 5.292M15 21H3v-1a6 6 0 0112 0v1zm0 0h6v-1a6 6 0 00-9-5.197M13 7a4 4 0 11-8 0 4 4 0 018 0z" />
                            </svg>
                        </a>
                    </nav>
                </div>
            </div>
        </div>
    </div>

```

```

        Team
        </a>

        <a href="#" class="text-gray-600 hover:bg-gray-50 hover:text-gray-900 group flex items-center px-2 py-2 text-sm font-medium rounded-md">
            <!-- Heroicon name: folder -->
            <svg class="text-gray-400 group-hover:text-gray-500 mr-3 h-6 w-6" xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0 24 24" stroke="currentColor" aria-hidden="true">
                <path stroke-linecap="round" stroke-linejoin="round" stroke-width="2" d="M3 7v10a2 2 0 002 2h14a2 2 0 002-2V9a2 2 0 00-2-2h-6l-2-2H5a2 2 0 00-2 2z" />
            </svg>
            Projects
            </a>

            <a href="#" class="text-gray-600 hover:bg-gray-50 hover:text-gray-900 group flex items-center px-2 py-2 text-sm font-medium rounded-md">
                <!-- Heroicon name: calendar -->
                <svg class="text-gray-400 group-hover:text-gray-500 mr-3 h-6 w-6" xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0 24 24" stroke="currentColor" aria-hidden="true">
                    <path stroke-linecap="round" stroke-linejoin="round" stroke-width="2" d="M8 7V3m8 4V3m-9 8h10M5 21h14a2 2 0 002-2V7a2 2 0 00-2-2H5a2 2 0 00-2 2v12a2 2 0 002 2z" />
                </svg>
                Calendar
                </a>

                <a href="#" class="text-gray-600 hover:bg-gray-50 hover:text-gray-900 group flex items-center px-2 py-2 text-sm font-medium rounded-md">
                    <!-- Heroicon name: inbox -->
                    <svg class="text-gray-400 group-hover:text-gray-500 mr-3 h-6 w-6" xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0 24 24" stroke="currentColor" aria-hidden="true">
                        <path stroke-linecap="round" stroke-linejoin="round" stroke-width="2" d="M20 13V6a2 2 0 00-2-2H6a2 2 0 00-2v7m16 0v5a2 2 0 01-2 2H6a2 2 0 01-2-2v-5m16 0h-2.586a1 1 0 00-.707.293l-2.414 2.414a1 1 0 01-.707.293h-3.172a1 1 0 01-.707-.293l-2.414-2.414a1 1 0 006.586 13H4" />
                    </svg>
                    Documents
                </a>
            </div>
        </div>
    </div>

```

```

        </a>

            <a href="#" class="text-gray-600 hover:bg-gray-50 hover:text-gray-900 group flex items-center px-2 py-2 text-sm font-medium rounded-md">
                <!-- Heroicon name: chart-bar -->
                <svg class="text-gray-400 group-hover:text-gray-500 mr-3 h-6 w-6" xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0 24 24" stroke="currentColor" aria-hidden="true">
                    <path stroke-linecap="round" stroke-linejoin="round" stroke-width="2" d="M9 19v-6a2 2 0 00-2-2H5a2 2 0 00-2 2v6a2 2 0 002 2h2a2 2 0 002-2zm0 0V9a2 2 0 012-2h2a2 2 0 012 2v10m-6 0a2 2 0 002 2h2a2 2 0 002-2m0 0V5a2 2 0 012-2h2a2 2 0 012 2v14a2 2 0 01-2h-2a2 2 0 01-2-2z" />
                </svg>
                Reports
            </a>
        </nav>
    </div>
</div>
</div>
</div>
</div>
</div>
<div class="flex flex-col w-0 flex-1 overflow-hidden">
    <div class="relative z-10 flex-shrink-0 flex h-16 bg-white shadow">
        <button class="px-4 border-r border-gray-200 text-gray-500 focus:outline-none focus:ring-2 focus:ring-inset focus:ring-indigo-500 md:hidden"
               @click.stop="openSidebar">
            <span class="sr-only">Open sidebar</span>
            <!-- Heroicon name: menu-alt-2 -->
            <svg class="h-6 w-6" xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0 24 24" stroke="currentColor" aria-hidden="true">
                <path stroke-linecap="round" stroke-linejoin="round" stroke-width="2" d="M4 6h16M4 12h16M4 18h7" />
            </svg>
        </button>
        <div class="flex-1 px-4 flex justify-between">
            <div class="flex-1 flex">
                </div>
            <div class="ml-4 flex items-center md:ml-6">
                <!-- Profile dropdown -->
            </div>
        </div>
    </div>

```

```

<div class="ml-3 relative"
      x-data="userPopupMenu()"
      @click.away="close"
      @keydown.window.escape="close">
    <div>
      <button class="max-w-xs bg-white flex items-center text-sm rounded-full focus:outline-none focus:ring-2 focus:ring-offset-2 focus:ring-indigo-500" id="user-menu" aria-haspopup="true"
              @click="toggleVisibility">
        <span class="sr-only">Open user menu</span>
        
      </button>
    </div>
    <!--
        Profile dropdown panel
    -->
    <div class="origin-top-right absolute right-0 mt-2 w-48 rounded-md shadow-lg py-1 bg-white ring-1 ring-black ring-opacity-5" role="menu" aria-orientation="vertical" aria-labelledby="user-menu"
        x-show="isVisible()"
        x-cloak
        x-transition:enter="transition ease-out duration-100 transform"
        x-transition:enter-start="opacity-0 scale-95"
        x-transition:enter-end="opacity-100 scale-100"
        x-transition:leave="transition ease-in duration-75 transform"
        x-transition:leave-start="opacity-100 scale-100"
        x-transition:leave-end="opacity-0 scale-95">
      <a href="#" class="block px-4 py-2 text-sm text-gray-700 hover:bg-gray-100" role="menuitem">Your Profile</a>
      <a href="#" class="block px-4 py-2 text-sm text-gray-700 hover:bg-gray-100" role="menuitem">Logout</a>
    </div>
  </div>

```

```

text-gray-700 hover:bg-gray-100" role="menuitem">Settings</a>
                                <a href="#" class="block px-4 py-2 text-sm
text-gray-700 hover:bg-gray-100" role="menuitem">Sign out</a>
                            </div>
                        </div>
                    </div>
                </div>
            </div>

        <main class="flex-1 relative overflow-y-auto focus:outline-none"
tabindex="0">
            <div class="py-6">
                <div class="max-w-7xl mx-auto px-4 sm:px-6 md:px-8">
                    <h1 class="text-2xl font-semibold text-gray-900">
Dashboard</h1>
                </div>
                <div class="max-w-7xl mx-auto px-4 sm:px-6 md:px-8">
                    <!-- Replace with your content -->
                    <div class="py-4">
                        <div class="border-4 border-dashed border-gray-
200 rounded-lg h-96"></div>
                    </div>
                    <!-- /End replace -->
                </div>
            </div>
        </main>
    </div>
</div>
<script src="https://unpkg.com/alpinejs@3.7.0/dist/cdn.min.js"
defer></script>
<script>

function sidebarMenu() {
    return {
        show: false,
        openSidebar() {
            this.show = true;
        },
        closeSidebar() {
            this.show = false;
        },
        isVisible() {
            return this.show === true;
        }
}

```

```

    };
}

function userPopupMenu() {
    return {
        show: false,
        toggleVisibility() {
            this.show = !this.show;
        },
        close() {
            this.show = false;
        },
        isVisible() {
            return this.show === true;
        }
    };
}
</script>
</body>
</html>

```

Try the application on different browser sizes and you will see that the left menu will appear or disappear as needed.

One final thing to do to have our application shell look good is adding our own logo.

#### 4.3.3. Serving static images

Spring Boot serves whatever files we put in `src/main/resources/public`. We can use this to serve the application logo. Create an `img` directory inside `src/main/resources/public` and put the logo there.

I took the Thymeleaf logo for now and named it `application-logo.png`.

Now we need to reference the logo in the `index.html`. Replace:

```

```

with:

```

```

This needs to be done in 2 places (One for the desktop sidebar, one for the mobile overlay).

The result should look like this on desktop:

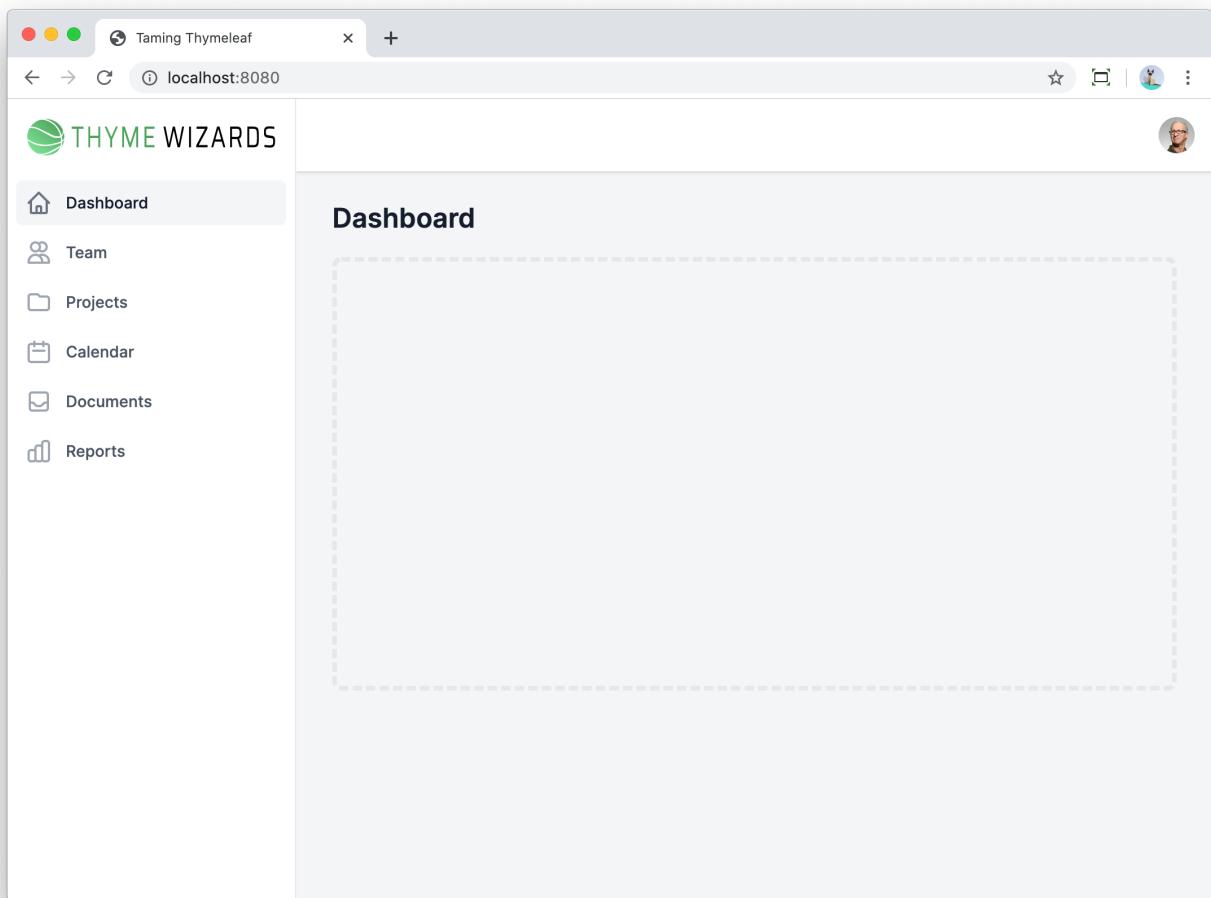


Figure 13. Application shell on desktop

and this on mobile:

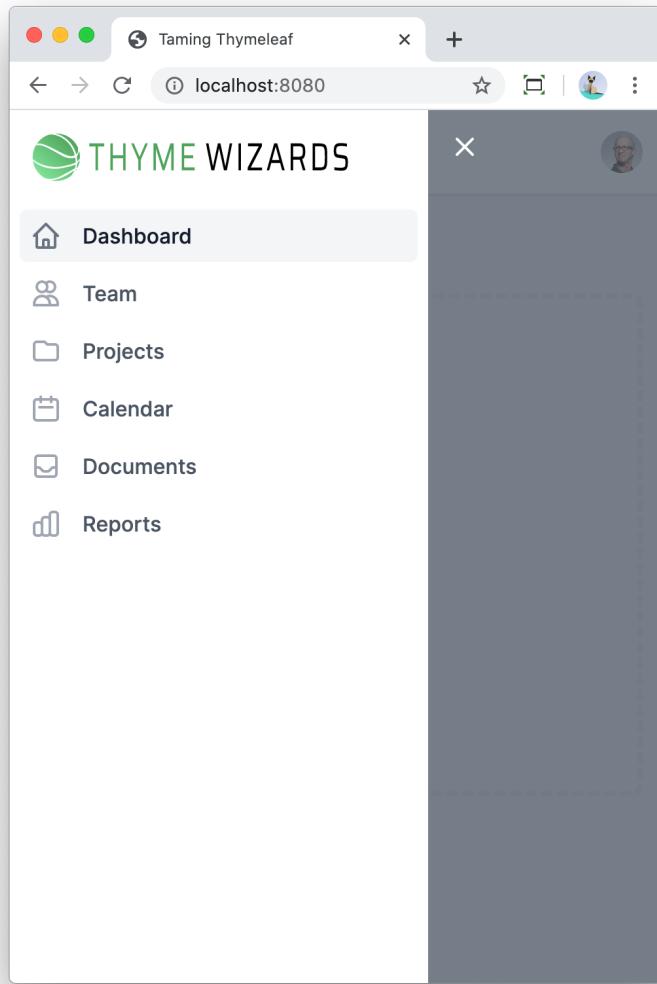


Figure 14. Application shell on mobile

## 4.4. Summary

In this chapter, you learned:

- Applying CSS in a Spring Boot Thymeleaf project
- Using Tailwind CSS to style the project
- Configuration of Tailwind CSS to build a custom design system
- Adding some interactivity on the page using AlpineJS.
- Serve and link to static images.

# Chapter 5. Fragments

Our `index.html` page has currently quite some duplication going on. If you are not familiar with Tailwind CSS, the many classes probably are the first thing you would tackle by creating custom CSS classes.

However, it is better to think in terms of full components that can be re-used. When we do that, there will be no need to define our own CSS classes to reduce the duplication.

## 5.1. What are fragments?

Thymeleaf has the concept of fragments, which are basically re-usable snippets of HTML. They are very similar to methods. Just as you use methods to better structure your Java code, you use fragments to better structure your HTML pages.

You define a fragment using `th:fragment`.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
    <body>
        <div th:fragment="separator"> ①
            <div class="border-dashed border-2 border-red-300 mx-4">
                </div>
        </div>
    </body>
</html>
```

① Defines a fragment with the `separator` name.

You can define a fragment on any HTML tag, it does *not need* to be a `<div>`.

## 5.2. Using fragments

We can now use this fragment in any Thymeleaf template. Suppose the above fragment is in a file called `fragments.html`. We can reference the specific `separator` fragment using this `~{filename :: fragmentname}` syntax:

```
<div>
    <div>There is some content here.</div>
    <div th:insert="~{fragments :: separator}"></div> ①
    <div>There is some more content here.</div>
</div>
```

① Use the `separator` fragment

By using `th:insert`, Thymeleaf will insert the content of the fragment as a child of the declared tag.

The `th:insert` attribute expects a fragment expression, which is defined by the `~{...}` syntax. However, it is also possible for non-complex fragment expression to leave out the `~{...}` part, so the example becomes:

```
<div>
    <div>There is some content here.</div>
    <div th:insert="fragments :: separator"></div>
    <div>There is some more content here.</div>
</div>
```

If we look at the resulting HTML, than it will look like this:

```
<div>There is some content here.</div>
<div> ①
    <div> ②
        <div class="border-dashed border-2 border-red-300 mx-4"> ③
            </div>
    </div>
</div>
<div>There is some more content here.</div>
```

- ① `<div>` from the "parent" HTML
- ② `div` from the fragment itself (The one that has the `th:fragment="separator"` attribute)
- ③ Child `div` from the fragment

To avoid the nested `<div>` tags, we can declare the fragment like this:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
    <body>
        <div th:fragment="separator" class="border-dashed border-2 border-red-300 mx-4"> ①
            </div>
    </body>
</html>
```

- ① `th:fragment` and `class` attributes on the same `<div>` tag

Using this fragment, the resulting HTML becomes:

```
<div>There is some content here.</div>
<div>
    <div class="border-dashed border-2 border-red-300 mx-4">
        </div>
```

```
</div>
<div>There is some more content here.</div>
```

It is perfectly possible to have attributes on the `<div>` (or any other HTML tag you want to use) that has the `th:fragment` declaration.

So far, we *inserted* the fragment, but we can also have Thymeleaf *replace* the tag from the "parent" document. To do that, use `th:replace` instead of `th:insert`:

```
<div>
    <div>There is some content here.</div>
    <div th:replace="fragments :: separator"></div> ①
    <div>There is some more content here.</div>
</div>
```

① Use the `separator` fragment with `th:replace`

The resulting HTML is now:

```
<div>There is some content here.</div>
<div class="border-dashed border-2 border-red-300 mx-4">
</div>
<div>There is some more content here.</div>
```

## 5.3. Fragments with parameters

Just like methods in Java code can have parameters, so can fragments in Thymeleaf have parameters.

As an example, we can imagine a menu item that is a fragment like this:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<body>
    <a th:fragment="menu-item(title, link)" ①
        th:text="${title}" ②
        th:href="${link}" ③
        class="flex items-center px-2 py-2 text-base leading-6 font-medium
text-gray-900">
    </a>
</body>
</html>
```

① Declare the `menu-item` fragment with 2 parameters: `title` and `link`

② Use the value of the `title` parameter as the link text

- ③ Use the value of the `link` parameter as the hyperlink reference

We can use this fragment for example like this:

```
<a th:replace="fragments :: menu-item('Users', '/users')"></a>
<a th:replace="fragments :: menu-item('Groups', '/groups')"></a>
```

Notice how there is really no need to create a custom CSS class for the classes that are the same over all menu items as we now have a fragment that has all the knowledge about how a menu item should look.

The resulting HTML will be this:

```
<a href="/users" class="flex items-center px-2 py-2 text-base leading-6 font-medium text-gray-900">Users</a>
<a href="/groups" class="flex items-center px-2 py-2 text-base leading-6 font-medium text-gray-900">Groups</a>
```



It is possible to declare a fragment without explicitly declaring the parameter names, but I would not recommend that as it makes it harder for users of the fragment to see what parameters are needed. See [Fragment local variables without fragment arguments](#) if you would like to use that.

If you want to make the parameters explicit on the calling side, that is possible like this:

```
<a th:replace="fragments :: menu-item(title='Users',
link='/users')"></a>
<a th:replace="fragments :: menu-item(title='Groups',
link='/groups')"></a>
```

When you do that, you can also change the order. So this would also give the same final result:

```
<a th:replace="fragments :: menu-item(link='/users',
title='Users')"></a>
<a th:replace="fragments :: menu-item(title='Groups',
link='/groups')"></a>
```

## 5.4. Fragments with HTML snippets as arguments

In our application, the menu items each have an SVG icon. As a reminder, here is how 1 menu item in our application looks like currently:

```
<a href="#" class="bg-gray-100 text-gray-900 group flex items-center px-2 py-2 text-sm font-medium rounded-md">
```

```

<!-- Current: "text-gray-500", Default: "text-gray-400 group-
hover:text-gray-500" -->
<!-- Heroicon name: home -->
<svg class="text-gray-500 mr-3 h-6 w-6"
xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0 24 24"
stroke="currentColor" aria-hidden="true">
<path stroke-linecap="round" stroke-linejoin="round" stroke-
width="2"
d="M3 12l2-2m0 0l7-7 7 7M5 10v10a1 1 0 001 1h3m10-11l2 2m-
2-2v10a1 1 0 01-1 1h-3m-6 0a1 1 0 001-1v-4a1 1 0 011-1h2a1 1 0 011 1v4a1
1 0 001 1m-6 0h6"/>
</svg>
Dashboard
</a>

```

If we want to create a parameterized fragment out of this, we need a way to pass the `<svg>` element content from the parent into the fragment.

We can do this as follows. Declare the fragment like this:

```

<a th:fragment="menu-item(title, link, svgContents)"①
th:href="${link}"
class="bg-gray-100 text-gray-900 group flex items-center px-2 py-2
text-sm font-medium rounded-md">
<svg th:replace="${svgContents}"></svg> ②
[[${title}]] ③
</a>

```

① Declare 3 parameters: `title`, `link` and `svgContents`

② Use the passed in `svgContents` as a child tag of the `<a>` tag

③ Have the `title` argument as the text of the menu item using [Expression inlining](#)



We cannot use `th:text` there because that completely replaces the body of the `<a>` tag with the text. If we did that, the SVG would not be visible in the rendered HTML.

We can now use this fragment like this:

```

<a th:replace="fragments :: menu-item('Dashboard', '/dashboard',
~{::#dashboard-icon})">
<svg id="dashboard-icon"
class="text-gray-500 mr-3 h-6 w-6"
fill="none" viewBox="0 0 24 24" stroke="currentColor">
<path stroke-linecap="round" stroke-linejoin="round" stroke-
width="2"

```

```

        d="M3 12l2-2m0 0l7-7 7 7M5 10v10a1 1 0 001 1h3m10-11l2 2m-
2-2v10a1 1 0 01-1 1h-3m-6 0a1 1 0 001-1v-4a1 1 0 011-1h2a1 1 0 011 1v4a1
1 0 001 1m-6 0h6"/>
</svg>
</a>
```

We gave the `<svg>` an `id` attribute of `dashboard-icon` so we can reference it when calling the fragment using the fragment expression `~{:::#dashboard-icon}`.

The resulting HTML:

```

<a href="/dashboard"
    class="bg-gray-100 text-gray-900 group flex items-center px-2 py-2
text-sm font-medium rounded-md">
    <svg id="dashboard-icon"
        class="text-gray-500 mr-3 h-6 w-6"
        fill="none" viewBox="0 0 24 24" stroke="currentColor">
        <path stroke-linecap="round" stroke-linejoin="round" stroke-
width="2"
            d="M3 12l2-2m0 0l7-7 7 7M5 10v10a1 1 0 001 1h3m10-11l2 2m-
2-2v10a1 1 0 01-1 1h-3m-6 0a1 1 0 001-1v-4a1 1 0 011-1h2a1 1 0 011 1v4a1
1 0 001 1m-6 0h6"></path>
    </svg>
    Dashboard
</a>
```

## 5.5. Inline separate SVG files

To inline SVG images, we can also use fragments. This can just work out of the box if we would name the file `my-icon.html` as Thymeleaf searches for `.html` files.

Better is that we can just use the `.svg` extension for them as we normally would. To make that possible, we do the following steps.

We create the `src/main/resources/templates/svg` directory and create the `dashboard.svg` file in there:

`src/main/resources/templates/svg/dashboard.svg`

```

<svg class="text-gray-500 mr-3 h-6 w-6"
xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0 24 24"
stroke="currentColor" aria-hidden="true">
    <path stroke-linecap="round" stroke-linejoin="round" stroke-
width="2" d="M3 12l2-2m0 0l7-7 7 7M5 10v10a1 1 0 001 1h3m10-11l2 2m-
2-2v10a1 1 0 01-1 1h-3m-6 0a1 1 0 001-1v-4a1 1 0 011-1h2a1 1 0 011 1v4a1
1 0 001 1m-6 0h6" />
```

```
</svg>
```

Next, we instruct Thymeleaf to search for fragments in the `svg` directory using the `.svg` suffix (as opposed to the default `.html` suffix). For this, we add the following Spring Boot configuration:

```
package com.tamingthymeleaf.application;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.thymeleaf.spring5.templateresolver.SpringResourceTemplateResolver;
import org.thymeleaf.templateresolver.ITemplateResolver;

@Configuration
public class TamingThymeleafApplicationConfiguration {

    @Bean
    public ITemplateResolver svgTemplateResolver() {
        SpringResourceTemplateResolver resolver = new
SpringResourceTemplateResolver();
        resolver.setPrefix("classpath:/templates/svg/");
        resolver.setSuffix(".svg");
        resolver.setTemplateMode("XML");

        return resolver;
    }
}
```

We also have to make sure that the default HTML template resolver has priority over our custom SVG resolver. For this, set the `spring.thymeleaf.template-resolver-order` property to `0` in `application.properties`:

`src/main/resources/application.properties`

```
# This ensures that the default HTML template resolver of Thymeleaf has
priority over our custom SVG resolver
spring.thymeleaf.template-resolver-order=0
```

If you get:



`java.io.FileNotFoundException: class path resource [templates/svg/index.svg] cannot be opened because it does not exist`

then you forgot to set the `spring.thymeleaf.template-resolver-order` property.

You can now use the SVG image like this:

```
<div>
    <svg th:replace="dashboard"></svg>
</div>
```

The resulting HTML will be:

```
<div>
    <svg class="text-gray-500 mr-3 h-6 w-6"
        xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0 24 24"
        stroke="currentColor" aria-hidden="true">
        <path stroke-linecap="round" stroke-linejoin="round" stroke-width="2"
            d="M3 12l2-2m0 0l7-7 7 7M5 10v10a1 1 0 001 1h3m10-11l2 2m-2-2v10a1 1 0 01-1 1h-3m-6 0a1 1 0 001-1v-4a1 1 0 011-1h2a1 1 0 011 1v4a1 1 0 001 1m-6 0h6"/>
    </svg>
</div>
```

So this approach has 2 advantages:

- You can easily re-use the SVG icon in multiple places.
- The file is a regular SVG file that can be viewed in image editors. When the SVG is embedded in the HTML, this is not possible.

*SVG as static content?*

You might wonder why we don't just add the SVG images [as static images](#), like we have done for the logo. The problem with using SVG images with an `<img>` tag, is that you cannot style them using CSS.

In many cases, we do want to style the icon to color it for example.



## 5.6. Homepage refactoring

Let's put our new knowledge to work and refactor the `index.html` into more manageable fragments.

We have the following parts in our index page:

- The mobile sidebar menu
- The desktop sidebar menu
- The top bar with the profile popup menu

We will add the mobile and desktop sidebar menu to `sidebar-menu.html` fragment and create `top-menu.html` for the top bar menu.

Move the relevant `<div>` section into the fragments and give them a name using `th:fragment`:

`src/main/resources/templates/fragments/sidebar-menu.html`

```
<html xmlns:th="http://www.thymeleaf.org">
<div th:fragment="mobile"
    class="md:hidden"
    x-show="isVisible()">
    ...
</div>
<div th:fragment="desktop" class="hidden md:flex md:flex-shrink-0">
    ...
</div>
</html>
```

Do the same for the `top-menu.html`:

`src/main/resources/templates/fragments/top-menu.html`

```
<html xmlns:th="http://www.thymeleaf.org">
<div th:fragment="menu" class="relative z-10 flex-shrink-0 flex h-16 bg-white shadow">
    ...
</div>
</html>
```

When we now use those fragments, our `index.html` becomes a lot more readable:

```
<div class="h-screen flex overflow-hidden bg-gray-100"
    x-data="sidebarMenu()"
    @keydown.window.escape="closeSidebar">
    <!-- Off-canvas menu for mobile -->
    <div th:replace="fragments/sidebar-menu :: mobile"></div> ①

    <!-- Static sidebar for desktop -->
    <div th:replace="fragments/sidebar-menu :: desktop"></div> ②

    <div class="flex flex-col w-0 flex-1 overflow-hidden">
        <div th:replace="fragments/top-menu :: menu"></div> ③

        <main class="flex-1 relative z-0 overflow-y-auto py-6
        focus:outline-none" tabindex="0">
            <div class="max-w-7xl mx-auto px-4 sm:px-6 md:px-8">
```

```

<h1 class="text-2xl font-semibold text-gray-900">
>Dashboard
    </h1>
</div>
<div class="max-w-7xl mx-auto px-4 sm:px-6 md:px-8">
    <!-- Replace with your content --&gt;
    &lt;div class="py-4"&gt;
        &lt;div class="border-4 border-dashed border-gray-200
rounded-lg h-96"&gt;
            &lt;div&gt;There is some content here.&lt;/div&gt;
        &lt;/div&gt;
    &lt;/div&gt;
    <!-- /End replace --&gt;
&lt;/div&gt;
&lt;/main&gt;
&lt;/div&gt;
&lt;/div&gt;
</pre>

```

- ① Use the `mobile` fragment from `sidebar-menu.html`
- ② Use the `desktop` fragment from `sidebar-menu.html`
- ③ Use the `menu` fragment from `top-menu.html`

We are not restricted to pure HTML in fragment, we can just as easily move our JavaScript into a Thymeleaf fragment.

Our `index.html` has currently this JavaScript for opening and closing the user menu when clicking the avatar:

```

function userPopupMenu() {
    return {
        show: false,
        toggleVisibility() {
            this.show = !this.show;
        },
        close() {
            this.show = false;
        },
        isVisible() {
            return this.show === true;
        }
    };
}

```

We can move this into the `top-menu.html` fragment like this:

```
<script th:fragment="user-popup-menu-js">
    function userPopupMenu() {
        return {
            show: false,
            toggleVisibility() {
                this.show = !this.show;
            },
            close() {
                this.show = false;
            },
            isVisible() {
                return this.show === true;
            }
        };
    }
</script>
```

And use it in the `index.html`:

```
<script src="https://unpkg.com/alpinejs@3.7.0/dist/cdn.min.js"
defer></script>
<script th:replace="fragments/top-menu :: user-popup-menu-js"></script>
```

At runtime, Thymeleaf will render this as:

```
<script src="https://unpkg.com/alpinejs@3.7.0/dist/cdn.min.js"
defer></script>
<script>
    function userPopupMenu() {
        return {
            show: false,
            toggleVisibility() {
                this.show = !this.show;
            },
            close() {
                this.show = false;
            },
            isVisible() {
                return this.show === true;
            }
        };
    }
</script>
```

```
</script>
```

Including JavaScript like this puts the JavaScript inside the actual HTML. This can be convenient at times, but there is an alternative that is normally used more often. Put the JavaScript into its own file and reference that.

To try that, create the file `src/main/resources/static/js/user-popup-menu.js` with this content:

```
function userPopupMenu() {
    return {
        show: false,
        toggleVisibility() {
            this.show = !this.show;
        },
        close() {
            this.show = false;
        },
        isVisible() {
            return this.show === true;
        }
    };
}
```

And reference it in the `index.html` like:

```
<script src="https://unpkg.com/alpinejs@3.7.0/dist/cdn.min.js"
defer></script>
<script th:src="@{/js/user-popup-menu.js}"></script>
```

Thymeleaf will render this as:

```
<script src="https://unpkg.com/alpinejs@3.7.0/dist/cdn.min.js"
defer></script>
<script src="/js/user-popup-menu.js"></script>
```

## 5.7. Menu item components

Now our `index.html` is already looking a lot better, but we still have a lot of duplication in our fragments themselves.

In the menu on the left side, each menu item looks similar to this:

```

<!-- Current: "bg-gray-100 text-gray-900", Default: "text-gray-600
hover:bg-gray-50 hover:text-gray-900" -->

    <!-- Current: "text-gray-500", Default: "text-gray-400 group-
hover:text-gray-500" -->
    <!-- Heroicon name: home -->
    <svg class="mr-3 h-6 w-6 text-gray-500 group-hover:text-gray-500
group-focus:text-gray-600 transition ease-in-out duration-150"
        fill="none" viewBox="0 0 24 24" stroke="currentColor">
        <path stroke-linecap="round" stroke-linejoin="round" stroke-
width="2"
            d="M3 12l2-2m0 0l7-7 7 7M5 10v10a1 1 0 001 1h3m10-11l2 2m-
2-2v10a1 1 0 01-1 1h-3m-6 0a1 1 0 001-1v-4a1 1 0 011-1h2a1 1 0 011 1v4a1
1 0 001 1m-6 0h6"/>
    </svg>
    Dashboard


```

We can identify the following structure:

- **<a>** HTML tag to identify this as a link that can be clicked to navigate to that part of the application
- **href** attribute that will indicate where to link to
- **class** attribute to style the menu item (NOTE: The comments indicate how the styles should change when the menu item is selected)
- **<svg>** child element with the relevant icon for the menu item
- **Dashboard** text that is shown to the user for the link

Let's create a fragment for this. Start with a file **sidebar-buttons.html** to put our fragment in:

*src/main/java/resources/templates/fragments/sidebar-buttons.html*

```

<html xmlns:th="http://www.thymeleaf.org">
<!--/*@thymesVar id="link" type="java.lang.String"*/-->
<!--/*@thymesVar id="title" type="java.lang.String"*/-->

</html>

```

This gives us the basic structure to work with. We declare 2 parameters to our fragment: **link** and **title**



The `@thymesVar` comment allows IntelliJ IDEA to know that there will be a variable with the given name and type available in the Thymeleaf context. Due to this, the editor can provide coding assistance.

To make sure our fragment already works, use it in `sidebar-menu.html`:

```
<a th:replace="fragments/sidebar-buttons :: desktop-button(link='#',  
title='Dashboard')"></a>
```

This will render as:

```
<a href="#">  
  Dashboard  
</a>
```

We will now add the `class` attribute. As the link can have 2 states (The menu item is selected, or it is not selected), we have to be careful how we do this.

The "Dashboard" menu item has the list of classes that should be used when a menu item is selected:

```
bg-gray-100 text-gray-900 group flex items-center px-2 py-2 text-sm  
font-medium rounded-md
```

We can look at any other menu item to check what classes are needed for an unselected item:

```
text-gray-600 hover:bg-gray-50 hover:text-gray-900 group flex items-  
center px-2 py-2 text-sm font-medium rounded-md
```

Looking closely, and based on the comments in the Tailwind UI template, these are the classes that are the same between those:

```
group flex items-center px-2 py-2 text-sm font-medium rounded-md
```

So we can apply this to our fragment:

`src/main/java/resources/templates/fragments/sidebar-buttons.html`

```
<html xmlns:th="http://www.thymeleaf.org">  
  <!--/*@thymesVar id="link" type="java.lang.String"-->  
  <!--/*@thymesVar id="title" type="java.lang.String"-->  
  <a th:fragment="desktop-button(link, title)"  
      th:href="${link}"  
      class="group flex items-center px-2 py-2 text-sm font-medium rounded-  
      md">
```

```
[[${title}]]  
</a>  
</html>
```

For the classes that are different, we will introduce another parameter to our fragment. We can call it `menuItem`. Later on, we will also ensure that there will be an `activeMenuItem` parameter in the Thymeleaf context so our component can correctly render itself.

To conditionally add CSS classes in Thymeleaf, we can use the `th:classappend` attribute. This will append to the list of classes that are present in the normal `class` attribute.

`src/main/java/resources/templates/fragments/sidebar-buttons.html`

```
<html xmlns:th="http://www.thymeleaf.org">  
<!--/*@thymesVar id="link" type="java.lang.String"*/--&gt;<br/><!--/*@thymesVar id="title" type="java.lang.String"*/--&gt;<br/><!--/*@thymesVar id="menuItem" type="java.lang.String"*/--&gt;<br/><!--/*@thymesVar id="activeMenuItem" type="java.lang.String"*/--&gt;<br/><a th:fragment="desktop-button(link, title, menuItem)"  
    th:href="${link}"  
    class="group flex items-center px-2 py-2 text-sm font-medium rounded-  
    md"  
    th:classappend="${activeMenuItem == menuItem}? 'bg-gray-100 text-  
    gray-900' : 'text-gray-600 hover:bg-gray-50 hover:text-gray-900'"  
>  
    [[${title}]]  
</a>  
  
</html>
```

We also have to pass in the new parameter where we use the fragment:

```
<a th:replace="fragments/sidebar-buttons :: desktop-button(link='#',  
    title='Dashboard', menuItem='dashboard')"></a>
```

Now that our CSS styling is ok, we need to look into adding the SVG icon. Since the SVG icons also need to be styled using CSS, we need to inline them.

We start with copying the `<svg></svg>` contents of each of the icons to their own files, removing the `class` attribute.

E.g. `dashboard.svg`:

`src/main/resources/templates/svg/dashboard.svg`

```
<svg class="text-gray-500 mr-3 h-6 w-6"  
    xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0 24 24"
```

```
stroke="currentColor" aria-hidden="true">
    <path stroke-linecap="round" stroke-linejoin="round" stroke-
width="2" d="M3 12l2-2m0 0l7-7 7 7M5 10v10a1 1 0 001 1h3m10-11l2 2m-2-
2v10a1 1 0 01-1 1h-3m-6 0a1 1 0 001-1v-4a1 1 0 011-1h2a1 1 0 011 1v4a1 1
0 001 1m-6 0h6" />
</svg>
```

Do the same for the other icons.

If you remember from [Update the CSS file](#), we created a `gulpfile.js` to be able to have live reloading while we are editing. This is configured to copy over HTML, CSS and JavaScript files, but not SVG. We will expand the build to also include the SVG files.

Add `copy-svg` and `copy-svg+css-and-reload` tasks to the `gulpfile.js` file:

```
gulp.task('copy-svg', () => gulp.src(['src/main/resources/**/*.{svg}'])
    .pipe(gulp.dest('target/classes/')));
gulp.task('copy-svg+css-and-reload', gulp.series('copy-svg', 'copy-css',
    reload));
```

Note that we also copy the CSS whenever the SVG changes, because SVG's can also be styled using the `class` attribute. When that happens, the Tailwind JIT compiler might need to generate new classes.

To ensure the Tailwind JIT compiler takes those SVG's into account, update the `tailwind.config.js` file:

```
const defaultTheme = require('tailwindcss/defaultTheme');

module.exports = {
    content: ['./src/main/resources/templates/**/*.{html,',
        './src/main/resources/templates/**/*.{svg}'], ①
    theme: {
        extend: {
            fontFamily: {
                sans: ['Inter var', ...defaultTheme.fontFamily.sans],
            },
        }
    },
    plugins: [
        require('@tailwindcss/forms')
    ]
};
```

① Add the SVG files as content for the Tailwind JIT compiler

Update the `build` task to use the `copy-svg` task:

```
gulp.task('build', gulp.series('copy-html', 'copy-svg', 'copy-css',
  'copy-js'));
```

Update the `watch` task to use the `copy-svg+css-and-reload` task:

```
gulp.task('watch', () => {
  browserSync.init({proxy: 'localhost:8080',});
  gulp.watch(['src/main/resources/**/*.{html}'], gulp.series('copy-
  html+css-and-reload'));
  gulp.watch(['src/main/resources/**/*.{svg}'], gulp.series('copy-
  svg+css-and-reload'));
  gulp.watch(['src/main/resources/**/*.{css}'], gulp.series('copy-css-
  and-reload'));
  gulp.watch(['src/main/resources/**/*.{js}'], gulp.series('copy-js-and-
  reload'));
});
```

Since we copy the SVG icons with gulp, we don't need Maven to copy them. Update `pom.xml` to also exclude them like we did for HTML, JS and CSS:

```
<resources>
  <resource>
    <directory>src/main/resources</directory>
    <excludes>
      <exclude>**/*.html</exclude>
      <exclude>**/*.css</exclude>
      <exclude>**/*.js</exclude>
      <exclude>**/*.svg</exclude>①
    </excludes>
  </resource>
</resources>
```

① Add `<exclude>` for SVG icons

After this, run the following to have live reloading working again:

```
npm run build && npm run watch
```

Let's go back to our icons now to finish up the styling of them.

Since we removed the `class` styling on the `<svg>` element itself, we will have to wrap our icons in a `<div>` to apply the same styling.

The styling for the icon when in a selected menu item is:

```
mr-3 h-6 w-6 text-gray-500
```

When not selected:

```
mr-3 h-6 w-6 text-gray-400 group-hover:text-gray-500
```

Again separating out the common styles like we did before, we end up with:

```
<a th:fragment="desktop-button(link, title, menuItem, icon)"
    th:href="${link}"
    class="mt-1 group flex items-center px-2 py-2 text-sm leading-5 font-medium rounded-md hover:text-gray-900 focus:outline-none transition ease-in-out duration-150"
    th:classappend="${activeMenuItem == menuItem}? 'text-gray-900 bg-gray-100 hover:bg-gray-100 focus:bg-gray-200' : 'text-gray-600 hover:bg-gray-50 focus:bg-gray-100'"
>
    <div class="mr-3 h-6 w-6"
        th:classappend="${activeMenuItem == menuItem}? 'text-gray-500' : 'text-gray-400 group-hover:text-gray-500' ">
        <svg th:replace="${icon}"></svg>
    </div>
    [[${title}]]
</a>
</html>
```

Note how we introduced a fourth parameter `icon` to pass in the name of the icon.

Our final snippet where we use the fragment becomes:

```
<a th:replace="fragments/sidebar-buttons :: desktop-button(link='#',
    title='Dashboard', menuItem='dashboard', icon='dashboard')"></a>
```



The parameter `icon` does not include the `.svg` file extension as our custom template resolver automatically adds the extension.

With this in place, we can replace the 50+ lines of HTML that makes up the menu for desktop, with the following:

```
<nav class="flex-1 px-2 bg-white">
    <a th:replace="fragments/sidebar-buttons :: desktop-button(link='#',
        title='Dashboard', menuItem='dashboard', icon='dashboard')"></a>
    <a th:replace="fragments/sidebar-buttons :: desktop-button(link='#',
```

```

title='Team', menuItem='team', icon='team')"></a>
    <a th:replace="fragments/sidebar-buttons :: desktop-button(link='#',
title='Projects', menuItem='projects', icon='projects')"></a>
    <a th:replace="fragments/sidebar-buttons :: desktop-button(link='#',
title='Calendar', menuItem='calendar', icon='calendar')"></a>
    <a th:replace="fragments/sidebar-buttons :: desktop-button(link='#',
title='Documents', menuItem='documents', icon='documents')"></a>
    <a th:replace="fragments/sidebar-buttons :: desktop-button(link='#',
title='Reports', menuItem='reports', icon='reports')"></a>
</nav>
```

Which is a lot more readable to say the least. Also, all the duplication of the CSS that we had, has disappeared by creating this re-usable Thymeleaf fragment.

All that is left now is do the same for the mobile menu items.

The fragment will be this:

```

<a th:fragment="mobile-button(link, title, menuItem, icon)"
    th:href="${link}"
    class="group flex items-center px-2 py-2 text-base font-medium
rounded-md"
    th:classappend="${activeMenuItem == menuItem} ? 'bg-gray-100 text-
gray-900' : 'text-gray-600 hover:bg-gray-50 hover:text-gray-900'"
>
    <div class="mr-4 h-6 w-6"
        th:classappend="${activeMenuItem == menuItem}?'text-gray-
500':'text-gray-400 group-hover:text-gray-500">
        <svg th:replace="${icon}"></svg>
    </div>
    [[${title}]]
</a>
```

And will be used in `sidebar-menu.html` like this:

```

<nav class="flex-1 px-2 bg-white">
    <a th:replace="fragments/sidebar-buttons :: desktop-button(link='#',
title='Dashboard', menuItem='dashboard', icon='dashboard')"></a>
    <a th:replace="fragments/sidebar-buttons :: desktop-button(link='#',
title='Team', menuItem='team', icon='team')"></a>
    <a th:replace="fragments/sidebar-buttons :: desktop-button(link='#',
title='Projects', menuItem='projects', icon='projects')"></a>
    <a th:replace="fragments/sidebar-buttons :: desktop-button(link='#',
title='Calendar', menuItem='calendar', icon='calendar')"></a>
    <a th:replace="fragments/sidebar-buttons :: desktop-button(link='#',
```

```
title='Documents', menuItem='documents', icon='documents')"></a>
    <a th:replace="fragments/sidebar-buttons :: desktop-button(link='#',
title='Reports', menuItem='reports', icon='reports')"></a>
</nav>
```

After all this, our application looks exactly the same to our users, but we have a much healthier code base to work on.

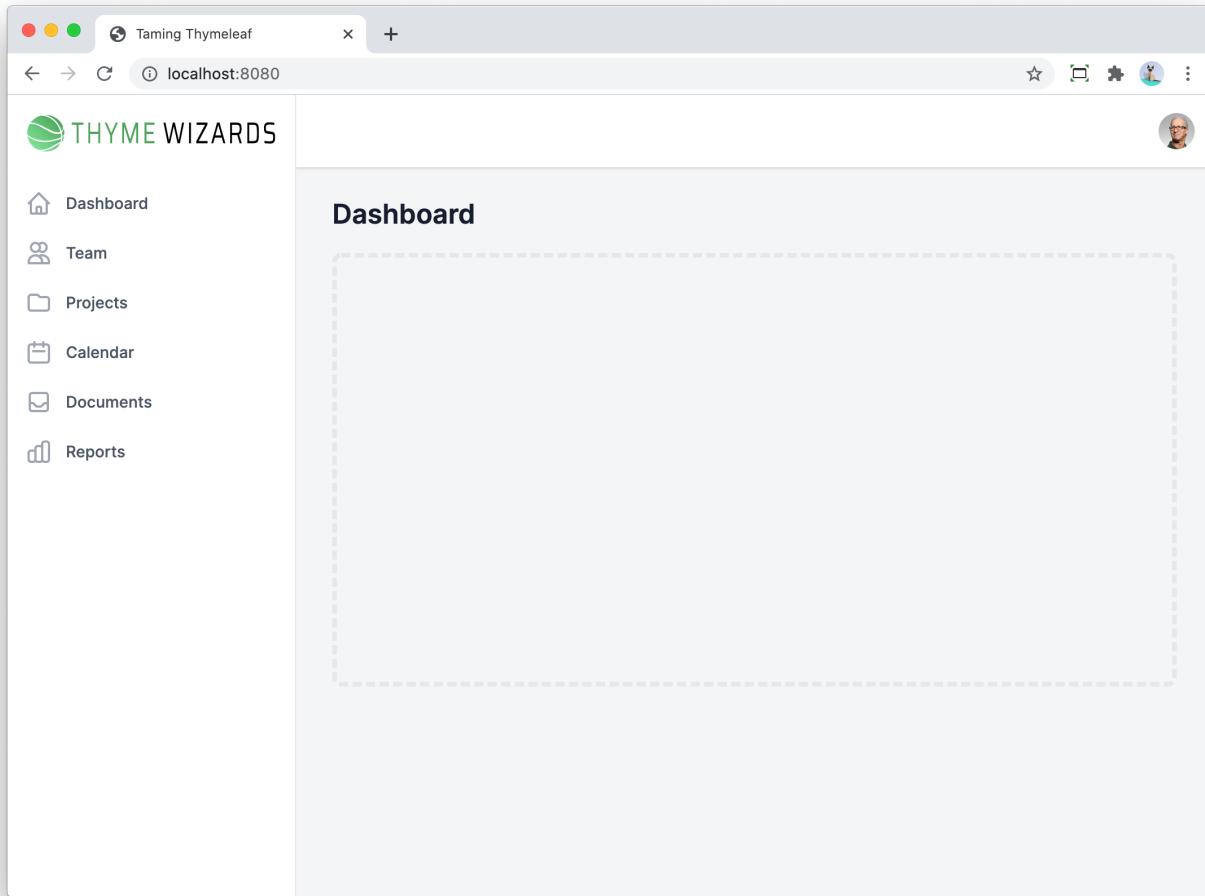


Figure 15. Application on desktop after fragments refactoring

## 5.8. Summary

In this chapter, you learned:

- What are fragments exactly and how can they be used
- How to use parameters in fragments
- Define custom fragment resolvers for non-HTML types of files (E.g. SVG)
- Applying the knowledge to refactor the home page into re-usable fragments.

# Chapter 6. Layouts

In the previous chapter, we explained how to use fragments to re-use parts of your HTML, or to just better structure the HTML for a clearer overview.

However, fragments cannot do everything.

If we start to think about all the pages the application will consist of, each page will need to display the side menu, the top menu, maybe a footer, and of course the central content that the page will be about. This can be a list of users, or the properties of a single user, or some other piece of information that might be editable or not.

Ideally, we define the template with the auxiliary content once and just add the content that is needed for the page in question. The *Thymeleaf Layout Dialect* allows us to do exactly that.

## 6.1. What is the Thymeleaf Layout Dialect?

The Thymeleaf Layout Dialect allows to build re-usable templates. You can define *extension points* so you can add content at arbitrary places in the template.

To use the Thymeleaf Layout Dialect, we need to add an extra dependency in the `pom.xml`:

`pom.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  https://maven.apache.org/xsd/maven-4.0.0.xsd">
  ...
  <dependencies>
    ...
    <dependency>
      <groupId>nz.net.ultraq.thymeleaf</groupId>
      <artifactId>thymeleaf-layout-dialect</artifactId>
    </dependency>
    ...
  </dependencies>
  ...
</project>
```

Let's look at an example:

`src/main/resources/templates/layout/layout.html`

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
  xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout">①
```

```

<head>
    <link rel="stylesheet" th:href="@{/css/application.css}">
</head>

<body>
<nav class="h-12 pl-4 bg-gray-100 shadow flex items-center justify-start">
    <a href="#" class="border-b-2 border-indigo-500 h-full inline-flex items-center">Menu Item 1</a>
    <a href="#" class="ml-6">Menu Item 2</a>
</nav>
<section layout:fragment="page-content" class="text-base text-gray-700 ml-4 mt-4"> ②

</section>
<script src="https://unpkg.com/alpinejs@3.7.0/dist/cdn.min.js"
defer></script>
<th:block layout:fragment="page-scripts">③
</th:block>
</body>
</html>

```

① Declare the `layout` namespace

② Use `layout:fragment` to define an extension point to the layout called `page-content`

③ Use a `th:block` as a layout fragment called `page-scripts`. A `th:block` tag itself is not rendered, but will render the content we put in the extension point as we use this template.

This layout we defined contains a minimal `<head>` section, a `<body>` with a menu and a `<section>` for the main content, as well as an extension point to add extra JavaScript.

We can use this layout as follows:

`src/main/resources/templates/index.html`

```

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      layout:decorate="~{layout/layout}">①
<head>
    <title>Thymeleaf Layout Dialect</title> ②
</head>
<body>
<section layout:fragment="page-content"> ③
    <div>Main content of the page goes here</div>
</section>
</body>

```

```
<th:block layout:fragment="page-scripts"> ④
    <script>
        function someFunction() {

    }
</script>
</th:block>
</html>
```

- ① Use `Layout:decorate` to indicate what layout should be used to decorate the current page. We use the fragment expression syntax (~) to link to the layout. The first `Layout` refers to the directory (relative to `src/main/resources/templates`) where the layout can be found. The second `Layout` refers to the file name (`layout.html`).
- ② Tags added to the `<head>` section will be merged by the dialect.
- ③ Use `layout:fragment` to indicate that you want to have the content of this `<section>` tag inserted at the `page-content` extension point of the layout.
- ④ Use the `page-scripts` extension point to add some JavaScript to the page.

When Thymeleaf renders this, the resulting HTML looks like this:

```
<!DOCTYPE html>
<html>
<head> ①
    <title>Thymeleaf Layout Dialect</title>
    <link rel="stylesheet" href="/css/application.css">
</head>

<body>
<nav class="h-12 pl-4 bg-gray-100 shadow flex items-center justify-start"> ②
    <a href="#" class="border-b-2 border-indigo-500 h-full inline-flex items-center">Menu Item 1</a>
    <a href="#" class="ml-6">Menu Item 2</a>
</nav>
<section class="text-base text-gray-700 ml-4 mt-4">
    <div>Main content of the page goes here</div> ③
</section>
<script src="https://unpkg.com/alpinejs@3.7.0/dist/cdn.min.js"
defer></script>
<script> ④
    function someFunction() {

    }
</script>
</body>
```

```
</html>
```

- ① The `<head>` contains the items from the layout and from the page
- ② The navigation comes from the layout
- ③ The content `<div>` is placed at the proper location
- ④ The Javascript is injected just before the `<body>` tag closes.

The browser finally renders this as:

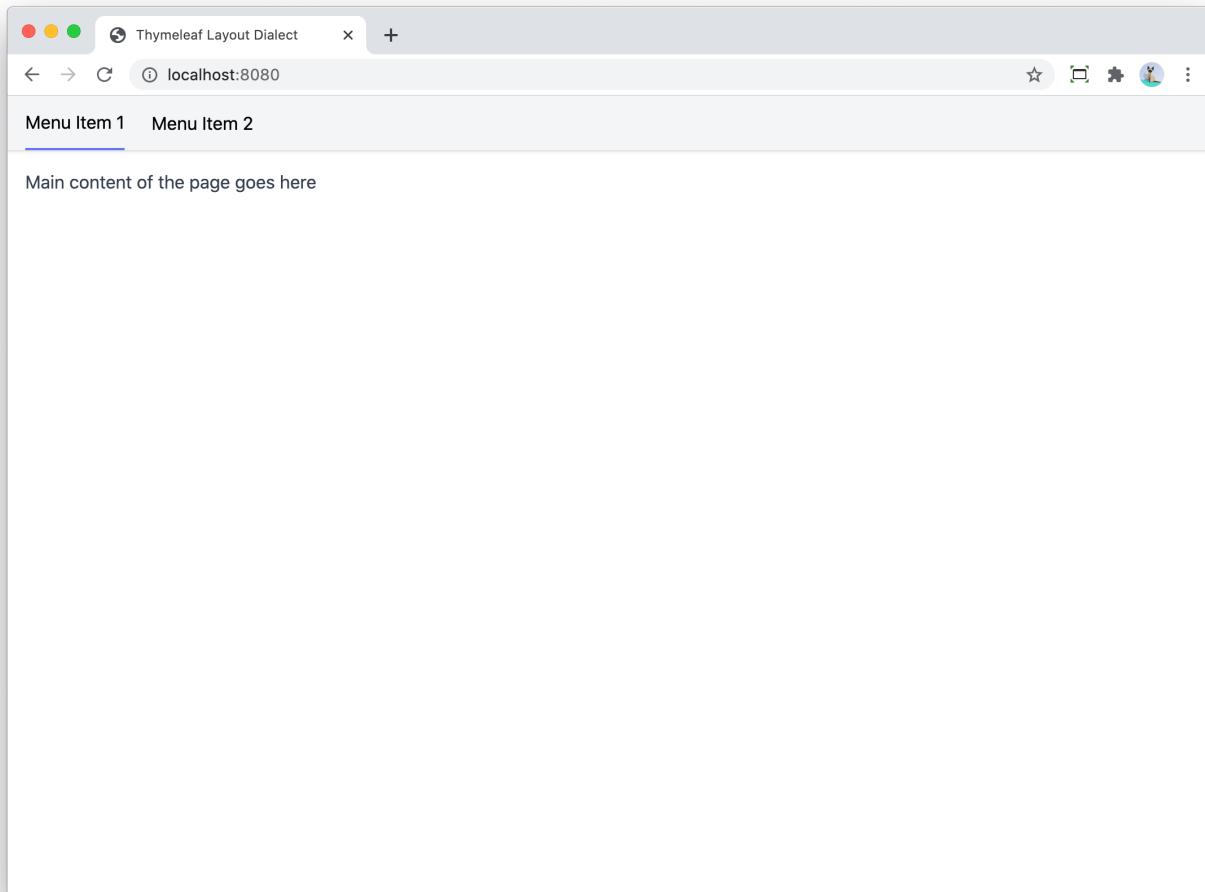


Figure 16. Rendering of layout example

## 6.2. Layouts with parameters

Layouts can have parameters, just like [fragments](#).

To show how this works, we will create an admonition fragment:

`src/main/resources/templates/layout/admonition.html`

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout">
<body>
```

```
<!--/*@thymesVar id="type" type="java.lang.String"-->
<div layout:fragment="admonition(type)" class="flex mb-4 p-2 w-2/4"
    th:classappend="${type == 'NOTE'?'bg-yellow-100':'bg-blue-100'}">
    <div th:text="${type}" class="mr-4"></div>
    <div layout:fragment="message"></div>
</div>
</body>
</html>
```

Since we don't want to use the full HTML page, but only the `<div>` and its children, we define a fragment name using `layout:fragment`. We also specify that the layout takes a `type` parameter. If the parameter has the value `NOTE`, we color the background yellow, otherwise, we color it blue.

We can use the template like this:

```
<section layout:fragment="page-content">
    <div class="mb-4">Main content of the page goes here</div>
    <div layout:replace="~{layout/admonition ::>
        admonition(type='NOTE')}"> ①
        <div layout:fragment="message"> ②
            This is an example note message.
        </div>
    </div>
    <div layout:replace="~{layout/admonition ::>
        admonition(type='TIP')}"> ③
        <div layout:fragment="message">
            You can use <span class="italic">parameters</span> with
            layouts.
        </div>
    </div>
</section>
```

① Use `layout:replace` to have this `<div>` we declare here replaced with the content of the admonition template. We pass in the `type` parameter with value `NOTE`.

② Our template has a `message` extension point where we can add any HTML content we like.

③ Pass in `TIP` as the value of the `type` parameter.

The resulting HTML will be:

```
<section class="text-base text-gray-700 ml-4 mt-4">
    <div class="mb-4">Main content of the page goes here</div>
    <div class="flex mb-4 p-2 w-2/4 bg-yellow-100">
        <div class="mr-4">NOTE</div>
        <div>
            This is an example note message.
        </div>
    </div>
```

```

        </div>
</div>
<div class="flex mb-4 p-2 w-2/4 bg-blue-100">
<div class="mr-4">TIP</div>
<div>
    You can use <span class="italic">parameters</span> with
    layouts.
</div>
</div>
</section>

```

When rendered in the browser, we get the following result:

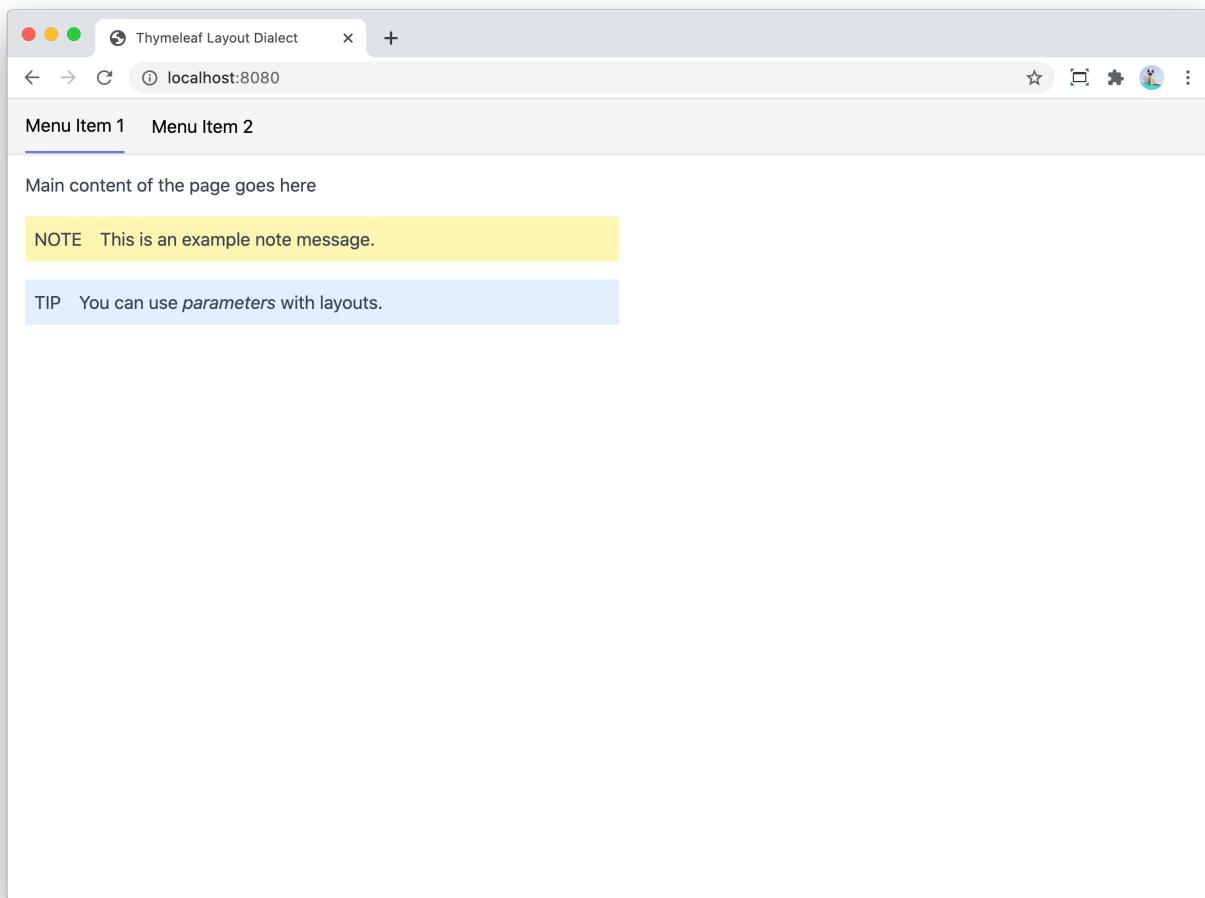


Figure 17. Using layouts with parameters

## 6.3. Page titles

The Thymeleaf Layout Dialect has special support for page titles. Suppose you want to have all pages to have a title that starts with `My Application` – and suffixed with something the page is about. E.g. `My Application - Users`, `My Application - Dashboard`, ...

If we define `<title>` each time in each page, we have to keep repeating the first part, making it hard

to change afterwards.

A better way to do this, is using `layout:title-pattern`. It allows to define the structure of the title at the layout level and add the correct suffix at the page level.

For example, suppose this is the `<head>` section of the layout:

```
<head>
    <title layout:title-pattern="$LAYOUT_TITLE - $CONTENT_TITLE">My
Application</title>
    <link rel="stylesheet" th:href="@{/css/application.css}">
</head>
```

Note the use of the special tokens: `$LAYOUT_TITLE` and `$CONTENT_TITLE`.

If we have a content page that uses the above layout like this:

```
<head>
    <title>Users</title>
</head>
```

This will result in the following HTML:

```
<head>
    <title>My Application - Users</title>
    <link rel="stylesheet" href="/css/application.css">
</head>
```

In this example, we used static text inside the `<title>` tag, but we can also use dynamic text with `th:text`.

## 6.4. Homepage refactoring

Let's apply this knowledge to our application. Start by creating a `layout.html` file in a `layout` package (relative to the `src/main/resources/templates` root).

`src/main/resources/templates/layout/layout.html`

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      lang="en">
<head>
    <meta charset="UTF-8">
```

```

<title layout:title-pattern="$LAYOUT_TITLE - $CONTENT_TITLE">Taming
Thymeleaf</title> ①
<meta http-equiv="X-UA-Compatible" content="IE=edge"/>
<meta name="viewport" content="width=device-width, initial-
scale=1"/>

<link rel="stylesheet" href="https://rsms.me/inter/inter.css">
<link rel="stylesheet" th:href="@{/css/application.css}">
</head>
<body>
<div class="h-screen flex overflow-hidden bg-gray-100"
    x-data="sidebarMenu()"
    @keydown.window.escape="closeSidebar">
    <!-- Off-canvas menu for mobile -->
    <div th:replace="fragments/sidebar-menu :: mobile"></div>

    <!-- Static sidebar for desktop -->
    <div th:replace="fragments/sidebar-menu :: desktop"></div>

    <div class="flex flex-col w-0 flex-1 overflow-hidden">
        <div th:replace="fragments/top-menu :: menu"></div>

        <main class="flex-1 relative z-0 overflow-y-auto py-6
focus:outline-none" tabindex="0">
            <div layout:fragment="page-content"> ②
                </div>
            </main>
        </div>
    </div>
<script src="https://unpkg.com/alpinejs@3.7.0/dist/cdn.min.js"
defer></script>
<script th:src="@{/js/user-popup-menu.js}"></script>
<script>
    function sidebarMenu() {
        return {
            show: false,
            openSidebar() {
                this.show = true;
            },
            closeSidebar() {
                this.show = false;
            },
            isVisible() {
                return this.show === true;
            }
        }
    }

```

```

    };
}
</script>
<th:block layout:fragment="page-scripts"> ③
</th:block>
</body>
</html>

```

① Set the `<title>` tag so that view using this layout can add to it

② Extension point for the main content of each view

③ Extension point for extra Javascript

We can now make the `index.html` as simple as:

`src/main/resources/templates/index.html`

```

<!DOCTYPE html>
<html
    xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
    layout:decorate="~{layout/layout}">
<head>
    <title>Dashboard</title>
</head>
<body>
<div layout:fragment="page-content">
    <div class="max-w-7xl mx-auto px-4 sm:px-6 md:px-8">
        <h1 class="text-2xl font-semibold text-gray-900">Dashboard</h1>
    </div>
    <div class="max-w-7xl mx-auto px-4 sm:px-6 md:px-8">
        <div class="py-4">
            <div class="border-4 border-dashed border-gray-200 rounded-lg h-96">
                <div></div>
            </div>
        </div>
    </div>
</div>
</body>
</html>

```

This will again visually in the browser won't change a thing, but it does set us up to quickly create other pages.

So far, we have been using `index.html` because Spring Boot will automatically serve this when present. In the next chapter, we will look into adding our own routes and adding more pages to our

application.

## 6.5. Summary

In this chapter, you learned:

- What are layouts and how can they be used.
- How to use parameters in layouts.
- Applying the knowledge to refactor the home page so it uses a layout.

# Chapter 7. Controllers

## 7.1. What is a controller?

Controllers in Spring MVC are the glue between the view (Thymeleaf templates) and the business logic. It is good practise to keep the controllers as small as possible. They should expose the business logic functionality over HTTP, not implement own logic.

Spring defines the `@Controller` annotation to mark a class as being a controller.

This is an example controller:

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

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

    @GetMapping("/all") ③
    public String index() {
        return "index"; ④
    }
}
```

- ① The `@Controller` annotation will make Spring find the class automatically and register it as a controller in the context.
- ② The `@RequestMapping` defines what path this controller controls. All `@GetMapping` annotated methods in this class will have an effective path that is relative to what is specified here.
- ③ The `@GetMapping` annotation indicates that a HTTP GET request to `/teams/all` will end up calling this `index()` method.
- ④ Methods in controllers can use a number of different return types. One of them is a plain `String`, which will be interpreted by Spring as the name of the Thymeleaf view to render (relative to `src/main/resources/templates` and without the `.html` extension of the file).

Each controller defines what path sections of the application it is responsible for. If there would be multiple controllers referring to the same path, then Spring will signal this and refuse to start:

```
java.lang.IllegalStateException: Ambiguous mapping. Cannot map
'otherTeamController' method
com.tamingthymeleaf.application.team.web.OtherTeamController#index(Model
)
to {GET /teams/all}: There is already 'teamController' bean method
```

Developers from a different background sometimes find this strange at first. They might be used to a single file containing all routes for the whole application. One could argue that you lack the overview of the routes, but in my experience, I never had a problem with that.

Note that different controllers can refer to the same path on a class level, but not on a method level.

This is possible as long as the resulting paths do not clash:

```
@Controller
@RequestMapping("/teams")
public class TeamController {

    @GetMapping("/all")
    public String listAll() {
        ...
    }

    @Controller
    @RequestMapping("/teams")
    public class TeamHistoryController {

        @GetMapping("/history")
        public String listHistory() {
            ...
        }
    }
}
```



Our controller method returned a `String` which is interpreted as the path to the Thymeleaf template to render. There are however many more return types possible. See [Handler Method Return Values](#) for the full details on what is possible.

## 7.2. Exposing data to the view

The method signature of controller methods is very flexible. Spring not only has dependency injection at the class level where you can use constructor injection (or field injection) to get references to dependent classes, but also at the method level.

One of the types that can be injected into a controller method is `org.springframework.ui.Model`. This class allows to add data (via the `addAttribute(String, Object)` method) that will be available to the view for rendering.

An example:

```
@Controller
@RequestMapping("/teams")
```

```

public class TeamController {

    private final TeamService service;

    public TeamController(TeamService service) { ①
        this.service = service;
    }

    @GetMapping("/all")
    public String index(Model model) { ②
        SortedSet<Team> teams = service.getTeams();
        model.addAttribute("teams", teams); ③

        return "teams/list";
    }
}

```

① Inject the service that has the business logic to get the set of teams.

② Add `Model` as method parameter. Spring will inject an instance of this class at runtime.

③ Add the set of teams under the `teams` key to the model.

We can now use that attribute in our Thymeleaf template like this:

```

<!DOCTYPE html>
<html
    xmlns:th="http://www.thymeleaf.org"
    xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
    layout:decorate="~{layout/layout}">
<head>
    <title>Teams</title>
</head>
<body>
<div layout:fragment="page-content">
    <div class="max-w-7xl mx-auto px-4 sm:px-6 md:px-8">
        <h1 class="text-2xl font-semibold text-gray-900">Teams</h1>
    </div>
    <div class="max-w-7xl mx-auto px-4 sm:px-6 md:px-8">
        <div class="py-4">
            <ol class="ml-4">
                <li th:each="team : ${teams}" class="list-disc"> ①
                    <span th:text="${team.name}"></span> ②
                </li>
            </ol>
        </div>
    </div>
</div>

```

```
</div>
</body>
</html>
```

- ① Use the exposed `teams` attribute to display the name of each team.
- ② We can call any method on the `Team` object to display information in the view. We could have used  `${team.getName()}` , but Thymeleaf also supports the shorter property notation  `${team.name}` .

Because we again used our layout, we immediately get a nicely rendered page with the menu and everything:

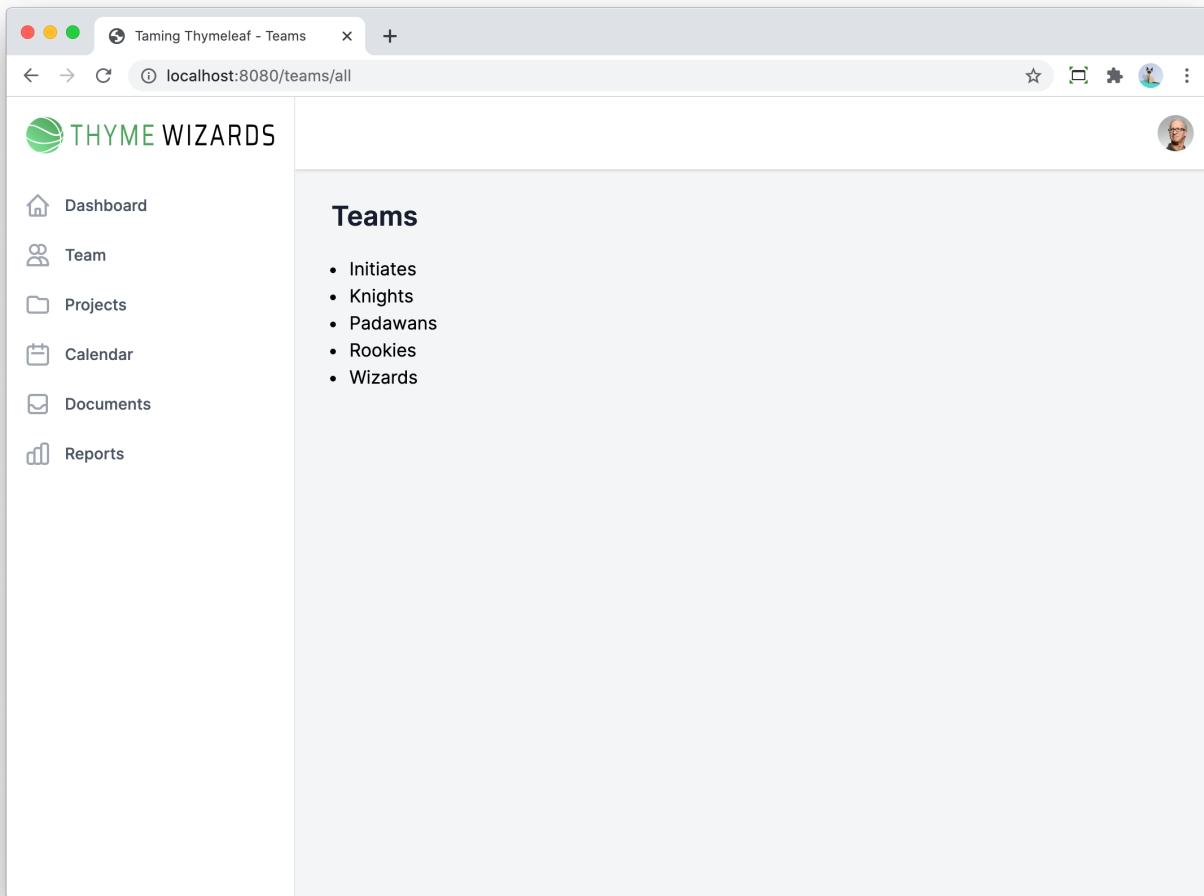


Figure 18. Using `@GetMapping` and `Model` to display the list of teams

Notice also that the title of the page is rendered as `Taming Thymeleaf - Teams` because of that `layout:title-pattern` that we used.

Request handler methods are very flexible in terms of what method arguments they can declare. You can view the full list at [Handler Methods Method Arguments](#). We will cover a few of those later on.

## 7.3. Path parameters

Our controller used a fixed, hardcoded path so far. But we can define very flexible mappings as well. One that you will need a lot is taking the id of something from the path to display information on that particular "thing".

In our example, we could show more information about each team, at `/teams/<id>` where `<id>` represents something that is unique about each team. In many cases, the primary key of the entity will be used, but that does not necessarily be the case.

```
@Controller
@RequestMapping("/teams")
public class TeamController {

    ...

    @GetMapping("/{id}") ①
    public String teamInfo(@PathVariable("id") String teamId, ②
                           Model model) {
        model.addAttribute("teamInfo", service.getTeamInfo(teamId)); ③
        return "teams/info";
    }
}
```

- ① Use the curly braces syntax to define that the part after `/teams/` should be captured for usage as a path variable.
- ② Have Spring inject the captured path variable as the `String teamId` into the controller method.
- ③ Use the `teamId` to get information about the team from the service and pass it as an attribute to the view.

It is also possible to have multiple path variables:

```
@Controller
@RequestMapping("/teams")
public class TeamController {

    ...

    @GetMapping("/{teamId}/players/{playerId}") ①
    public String playerOnTeamInfo(@PathVariable("teamId") String
teamId,
                                   @PathVariable("playerId") String
playerId, ②
                                   Model model) {
        model.addAttribute("player", service.getPlayerOnTeam(teamId,
playerId));
        return "teams/info";
    }
}
```

- ① Define multiple path variables using the curly brace syntax.
- ② Add a `@PathVariable` annotated method argument for each path variable.

## 7.4. Posting data

We saw how to use `@GetMapping` to display information to the user. Inevitably, the user will also want to change data. In web terms, this is done using HTTP `POST` with a `<form>`.

Suppose we have a form to change the name of a team. The controller method to make that possible could look something like this:

```
@Controller
@RequestMapping("/teams")
public class TeamController {
    ...
    @PostMapping("/{id}") ①
    public String editTeamName(@PathVariable("id") String teamId,
                               @ModelAttribute("editTeamFormData")
                               EditTeamFormData formData) { ②

        service.changeTeamName(teamId, formData.getTeamName()); ③

        return "redirect:/teams/all"; ④
    }
}
```

- ① Use the `@PostMapping` annotation to indicate that a `POST` call to `/teams/<id>` should be handled by this method.
- ② `EditTeamFormData` is a simple POJO that you need to create to match the fields of the form you are POST'ing.
- ③ Use the data to update the team name.
- ④ By using `redirect:` in the returned String, we instruct Spring to redirect to another page after the POST. This is a pattern called `Post/Redirect/Get` that is used a lot in web development. By redirecting, you avoid that the `POST` could be submitted twice if the user would refresh.

This is the basics of a `@PostMapping`. We will go into more detail later about how the form and the form data Java object exactly should match up. We will also learn about proper error handling as this example is lacking that for the moment.

## 7.5. Support for other HTTP methods

Next to `@GetMapping` and `@PostMapping`, there are also dedicated annotations for the other HTTP methods.

This table shows the full list:

HTTP method	Spring MVC annotation
GET	<code>@GetMapping</code>

HTTP method	Spring MVC annotation
POST	@PostMapping
PUT	@PutMapping
PATCH	@PatchMapping
DELETE	@DeleteMapping



There is also the general `@RequestMapping` annotation that requires a parameter to instruct what HTTP method should be used. For example: `@RequestMapping(method = RequestMethod.POST)` is equivalent to `@PostMapping`. Most new code will prefer the shorter forms, but now you know about it should you run into it one day.

All these HTTP verbs are nice, but web browsers only support `GET` and `POST`. If you want to use the other 3 as well in your web application, you need to use the `org.springframework.web.filter.HiddenHttpMethodFilter`. You enable this in Spring Boot by setting the following property:

`src/main/resources/application.properties`

```
spring.mvc.hiddenmethod.filter.enabled=true
```

This allows to add a hidden input field in the form named `_method` that contains the wanted HTTP method (`PUT`, `PATCH` or `DELETE`)

[Chapter 13](#) will explain this in more detail and show an example of using this with the `DELETE` HTTP method.

## 7.6. Team and User controllers

Let's apply our newly found knowledge to the application we are building. We will start with creating 2 controllers:

- `com.tamingthymeleaf.application.user.web.UserController`: This controller is responsible for the users of the application. In our example, these are the basketball players, the coaches, the administrators, etc...
- `com.tamingthymeleaf.application.team.web.TeamController`: This controller is responsible for the teams within the application.



A "team" in this application is a group of players of the Thyme Wizards playing together. For the children, this is an age group. For older players, this is based on merit.

This is the code for the `UserController`:

`com.tamingthymeleaf.application.user.web.UserController`

```
package com.tamingthymeleaf.application.user.web;
```

```

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

@Controller
@RequestMapping("/users")
public class UserController {

    @GetMapping
    public String index(Model model) {
        return "users/list";
    }
}

```

And the **TeamController** is very similar:

*com.tamingthymeleaf.application.team.web.TeamController*

```

package com.tamingthymeleaf.application.team.web;

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

@Controller
@RequestMapping("/teams")
public class TeamController {

    @GetMapping
    public String index(Model model) {
        return "teams/list";
    }
}

```

We also need 2 views:

- `templates/users/list.html`
- `templates/teams/list.html`

This is the source listing for the `users/list.html` view, but the other one is almost the same except for the text in the title.

src/main/resources/templates/users/list.html

```
<!DOCTYPE html>
<html
    xmlns:th="http://www.thymeleaf.org"
    xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
    layout:decorate="~{layout/layout}">

<head>
    <title>Users</title>
</head>
<body>
<div layout:fragment="page-content">
    <div class="max-w-7xl mx-auto px-4 sm:px-6 md:px-8">
        <h1 class="text-2xl font-semibold text-gray-900">Users</h1>
    </div>
    <div class="max-w-7xl mx-auto px-4 sm:px-6 md:px-8">
        <div class="py-4">
            <span>TODO show list of users</span>
        </div>
    </div>
</div>
</body>
</html>
```

Next, we change the menu items to have "Users" and "Teams". For the icon, we can use the icons from the [Heroicons](#) set, as that is what the Tailwind UI application shell is using already.

Copy the `users.svg` and the `user-group.svg` icons from the website into the `src/main/resources/templates/svg` directory.

Adjust the `templates/fragments/sidebar-menu.html` to use the icons and change the name of the menu items. This needs to be done for the mobile menu and the desktop menu. As an example, this is the desktop menu code:

```
<nav class="flex-1 px-2 bg-white">
    <a th:replace="fragments/sidebar-buttons :: desktop-button(link='#', title='Dashboard', menuItem='dashboard', icon='dashboard')"></a>
    <a th:replace="fragments/sidebar-buttons :: desktop-button(link=@{/users}, title='Users', menuItem='users', icon='users')"></a> ①
    <a th:replace="fragments/sidebar-buttons :: desktop-button(link=@{/teams}, title='Teams', menuItem='teams', icon='user-group')"></a> ②
    <a th:replace="fragments/sidebar-buttons :: desktop-button(link='#', title='Calendar', menuItem='calendar', icon='calendar')"></a>
    <a th:replace="fragments/sidebar-buttons :: desktop-button(link='#', title='Calendar', menuItem='calendar', icon='calendar')"></a>
```

```
title='Documents', menuItem='documents', icon='documents')"></a>
<a th:replace="fragments/sidebar-buttons :: desktop-button(link='#',
title='Reports', menuItem='reports', icon='reports')"></a>
</nav>
```

- ① The "Users" menu link. The `link` parameter now refers to our `UserController` via the `@{/users}` value.
- ② The "Teams" menu link referring to the `TeamController` via `@{/teams}`.

With this in place, we can navigate between both pages using the desktop or mobile menu:

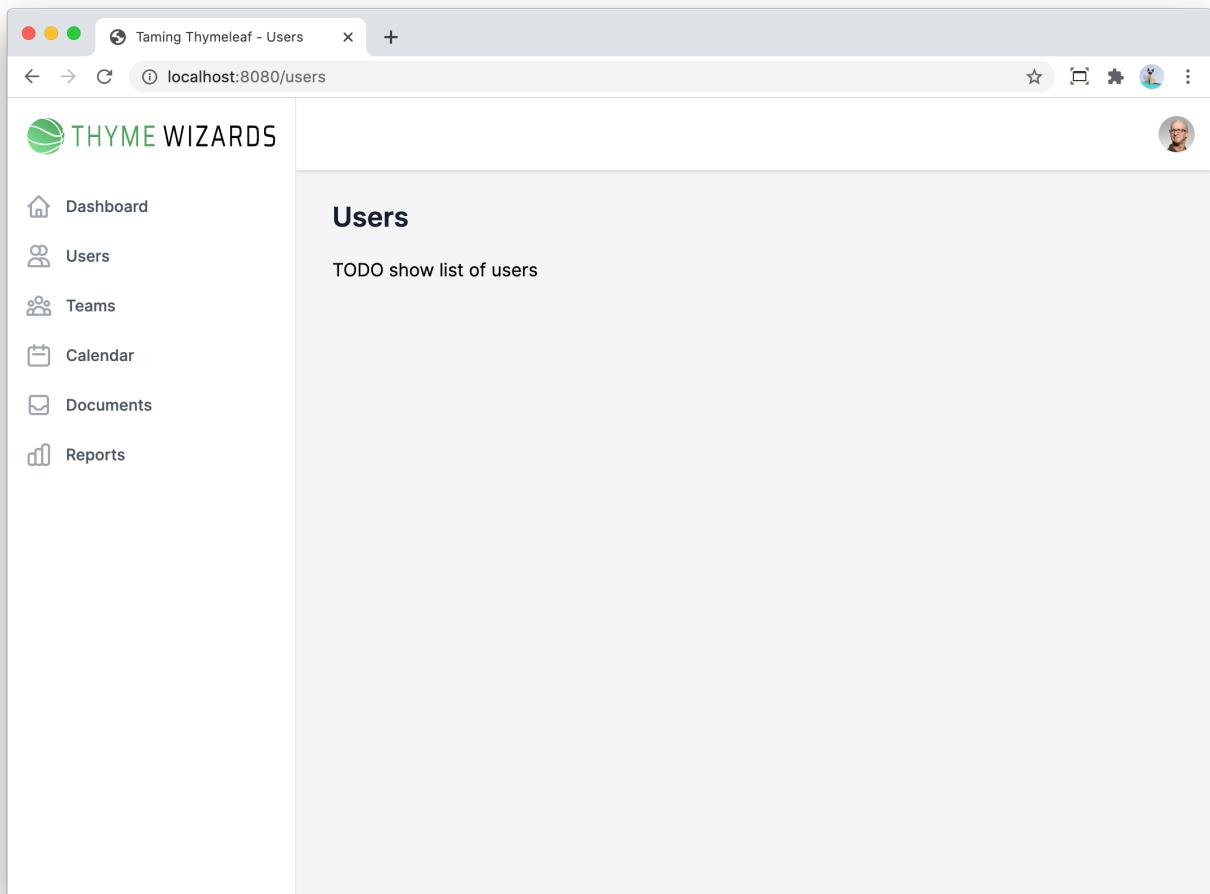


Figure 19. The `/users` page

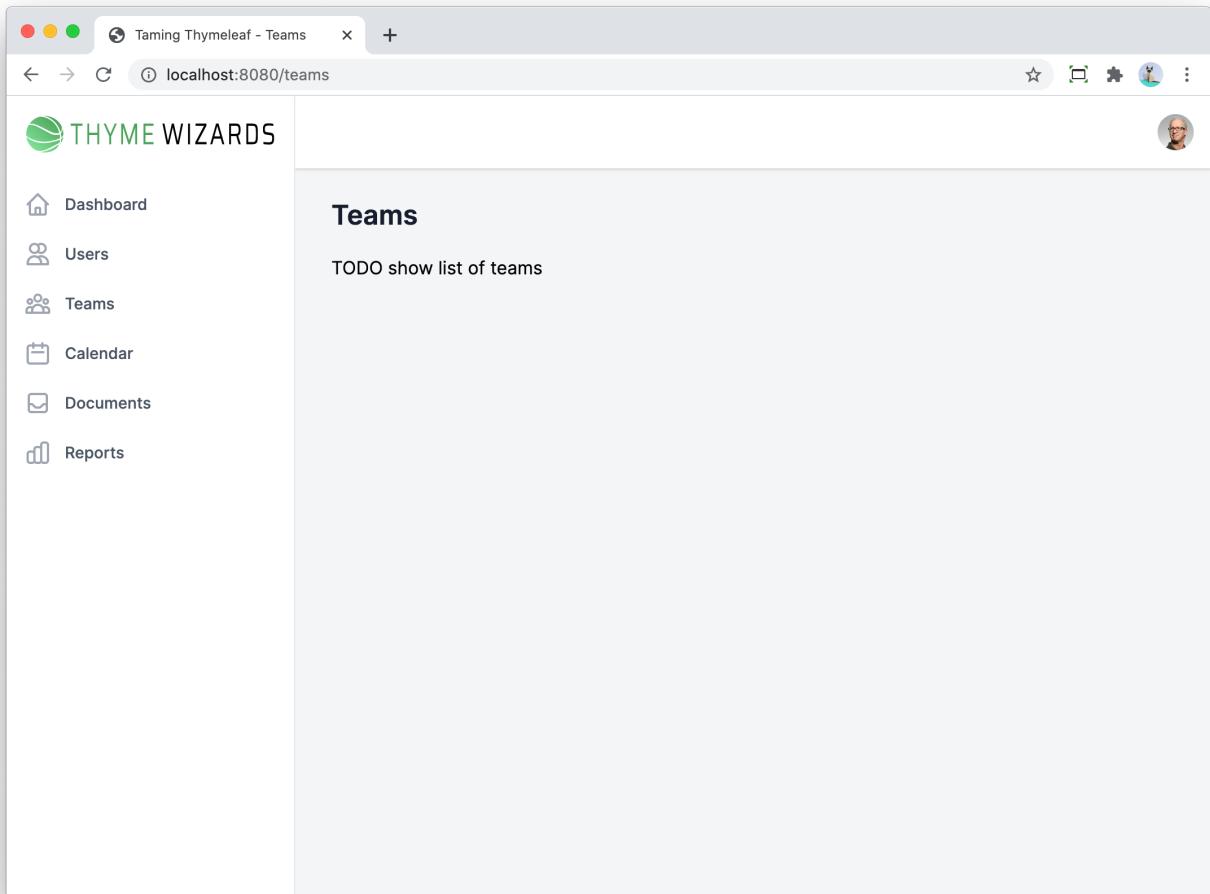
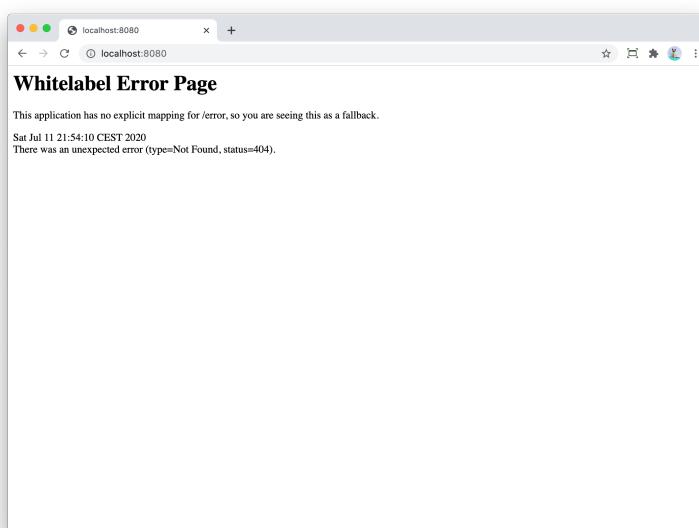


Figure 20. The `/teams` page

Going to <http://localhost:8080> (or <http://localhost:3000> if the live reload is active) will now result in an error:



This is because the default controller that served the `index.html` is no longer active as we now started adding our own controllers. Be sure to go to <http://localhost:8080/>

[users](#) or <http://localhost:8080/teams> directly when testing this.

When looking at the screenshots, it becomes clear that we are missing an indication in the menu of which menu item is currently selected.

If you remember from the chapter on fragments, we included an `activeMenuItem` property in our fragment:

```
<a th:fragment="desktop-button(link, title, menuItem, icon)"
    th:href="${link}"
    class="group flex items-center px-2 py-2 text-sm font-medium rounded-md"
    th:classappend="${activeMenuItem == menuItem} ? 'bg-gray-100 text-gray-900' : 'text-gray-600 hover:bg-gray-50 hover:text-gray-900'"
>
    <div class="mr-3 h-6 w-6"
        th:classappend="${activeMenuItem == menuItem}?'text-gray-500':'text-gray-400 group-hover:text-gray-500">
        <svg th:replace="${icon}"></svg>
    </div>
    [[${title}]]
</a>
```

We can now set this property in our view by using `th:with`:

```
<!DOCTYPE html>
<html
    xmlns:th="http://www.thymeleaf.org"
    xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
    layout:decorate="~{layout/layout}"
    th:with="activeMenuItem='users'"> ①
<head>
    <title>Users</title>
</head>
<body>
<div layout:fragment="page-content">
    <div class="max-w-7xl mx-auto px-4 sm:px-6 md:px-8">
        <h1 class="text-2xl font-semibold text-gray-900">Users</h1>
    </div>
    <div class="max-w-7xl mx-auto px-4 sm:px-6 md:px-8">
        <div class="py-4">
            <span>TODO show list of users</span>
        </div>
    </div>
</div>
</body>
```

```
</body>
</html>
```

① Set the `activeMenuItem` property to `users`

For reference, here is the menu item code for the "Users" menu:

```
<a th:replace="fragments/sidebar-buttons :: desktop-
button(link=@{/users}, title='Users', menuItem='users',
icon='users')"></a>
```

Because the value of `menuItem` of our `desktop-button` now matches with the value of the `activeMenuItem` (set using `th:with` on the page itself), the "Users" menu will be shown highlighted:

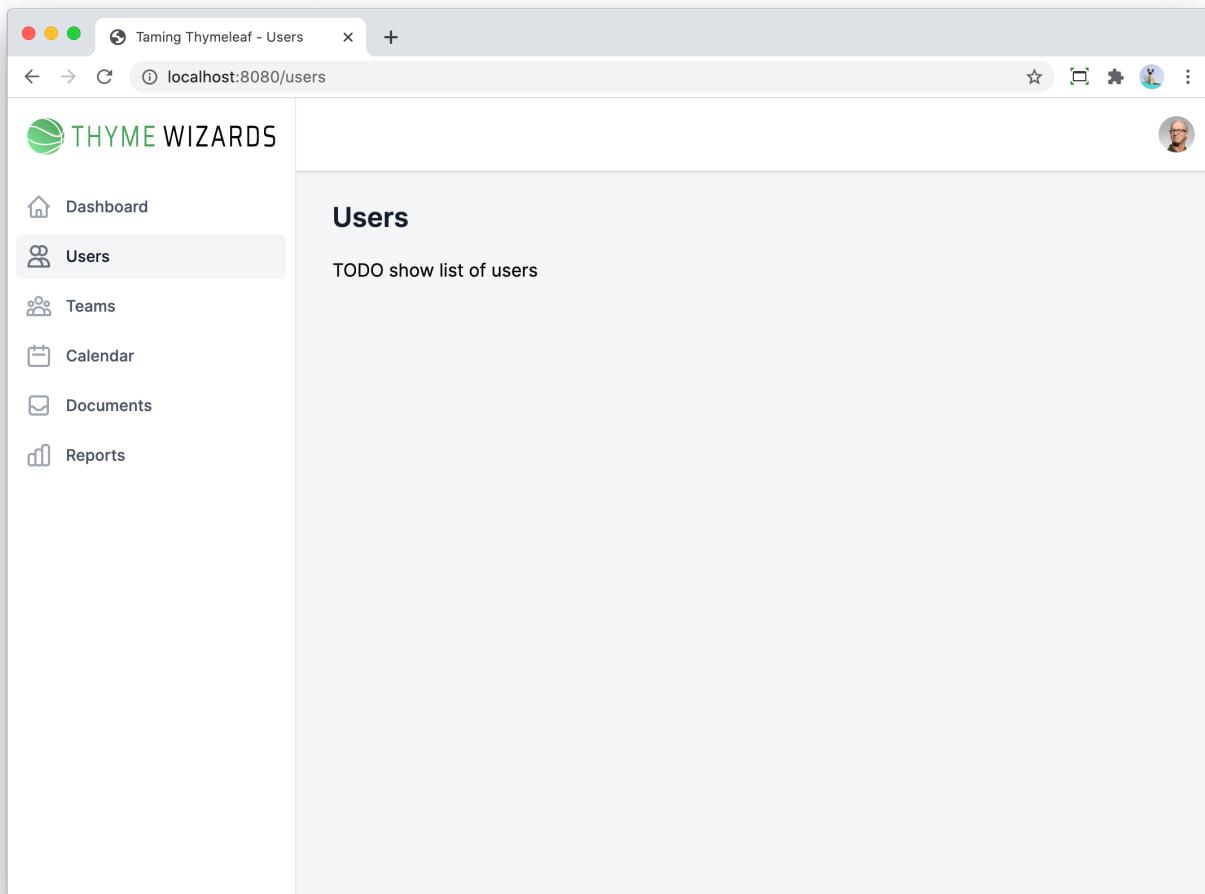


Figure 22. The Users page with the menu item highlighted

As a last thing before we close down this chapter, we will fix the problem that the root url (`http://localhost:8080`) is no longer working. Since we only have the Users and the Teams pages, we can redirect to either one of them. Let's implement a redirect from the root to `/users` as an example.

Add a `RootController` class:

```
package com.tamingthymeleaf.application.infrastructure.web;
```

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

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

    @GetMapping
    public String root() {
        return "redirect:/users"; ②
    }
}
```

① Map the controller to the root path

② Redirect to /users

If you now go to <http://localhost:8080>, you will get redirected automatically to <http://localhost:8080/users>.

## 7.7. Summary

In this chapter, you learned:

- What are controllers and how are they linked to the routes the application exposes
- The different request mapping annotations, and the link to the corresponding HTTP methods
- How to implement highlighting the active menu item

# Chapter 8. Internationalization

I live in Belgium where we have 3 official languages. It might be the reason I find internationalization (or i18n as it is called sometimes) important.

Using Spring Boot and Thymeleaf, it is really not that difficult to support multiple languages.

## 8.1. Internationalization basics

By default, Spring Boot expects a properties file `messages.properties` at the root of the classpath, so that would be at `src/main/resources` in the default directory structure. For each language the application supports, another file is added at that same location. I like it better to have all those translation files in their own directory. Add the following property to `application.properties` to accomplish this:

```
spring.messages.basename=i18n/messages
```

Now create `src/main/resources/i18n/messages.properties`:

```
users.title=Users
```

This file has to contain a key and a value for each word or sentence that needs to be translated.

We can now refer to the translated text by using the key with the `#{}...` syntax in the `th:text` attribute:

```
<div class="max-w-7xl mx-auto px-4 sm:px-6 md:px-8">
    <h1 class="text-2xl font-semibold text-gray-900"
        th:text="#{users.title}">Users</h1>
</div>
```

Let's add another language, for example Dutch. The ISO 639-1 language code for Dutch is `nl`, so we should create a file called `messages_nl.properties`:

```
users.title=Gebruikers
```

We use the same `users.title` key again here, but with the Dutch translation this time.



It is also possible to add a country code variant to the name of the messages file. Use `messages_nl_BE.properties` and `messages_nl_NL.properties` if you'd like to use different translations for people from Belgium speaking Dutch, as compared to people from the Netherlands speaking Dutch.

See [List of ISO 3166 country codes](#) for the full list of possible country codes.

To test this in Chrome, you can use the [Advanced Page Language Switcher](#) Chrome extension. This extension changes the `Accept-Language` header that the browser sends out, so our application will return the translated page:

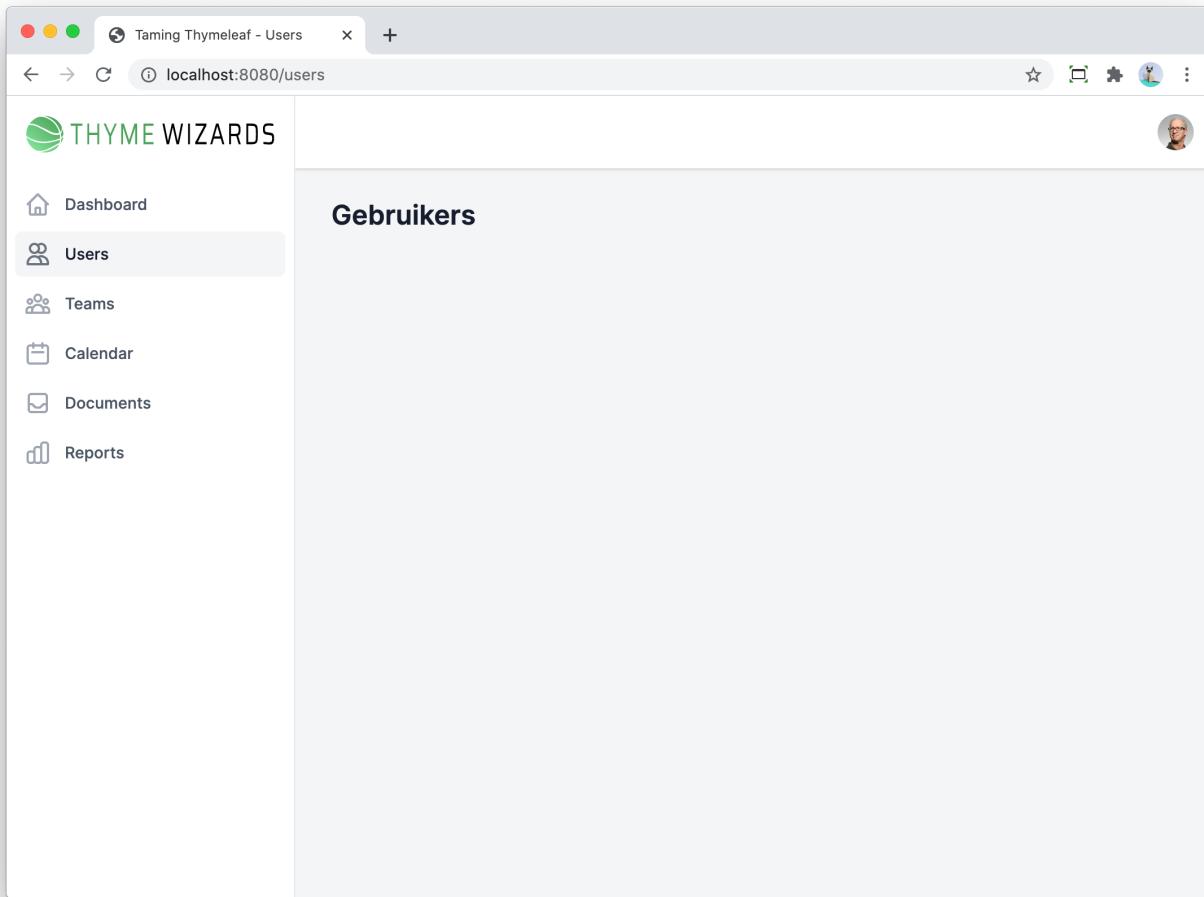


Figure 23. Page title translated to Dutch

## 8.2. Using a query parameter to select the language

The default browser language support is nice, but sometimes, you might want to give the user more control to select the wanted language.

We can add a query parameter that will set the language by creating a class that implements `org.springframework.web.servlet.config.annotation.WebMvcConfigurer` with a `LocaleChangeInterceptor`:

```
package com.tamingthymeleaf.application.infrastructure.web;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.LocaleResolver;
import
org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import
```

```

org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
import org.springframework.web.servlet.i18n.CookieLocaleResolver;
import org.springframework.web.servlet.i18n.LocaleChangeInterceptor;

@Configuration ①
public class WebMvcConfiguration implements WebMvcConfigurer { ②
    @Bean
    public LocaleResolver localeResolver() {
        return new CookieLocaleResolver(); ③
    }

    @Bean
    public LocaleChangeInterceptor localeInterceptor() { ④
        LocaleChangeInterceptor localeInterceptor = new
        LocaleChangeInterceptor();
        localeInterceptor.setParamName("lang");
        return localeInterceptor;
    }

    @Override
    public void addInterceptors(InterceptorRegistry registry) { ⑤
        registry.addInterceptor(localeInterceptor());
    }
}

```

① `@Configuration` ensures Spring Boot will find this class when scanning.

② Implement the `org.springframework.web.servlet.config.annotation.WebMvcConfigurer` interface that defines all callback methods that can be used to configure the Spring Web MVC setup.

③ Create a `CookieLocaleResolver` to store the selected language in a browser cookie.

④ Create a `LocaleChangeInterceptor` and configure the parameter name to use.

⑤ Add the interceptor to the registry of interceptors.

We can now open <http://localhost:8080/users?lang=nl> in the browser and the Dutch translation will be used (although the default language of the browser is English):

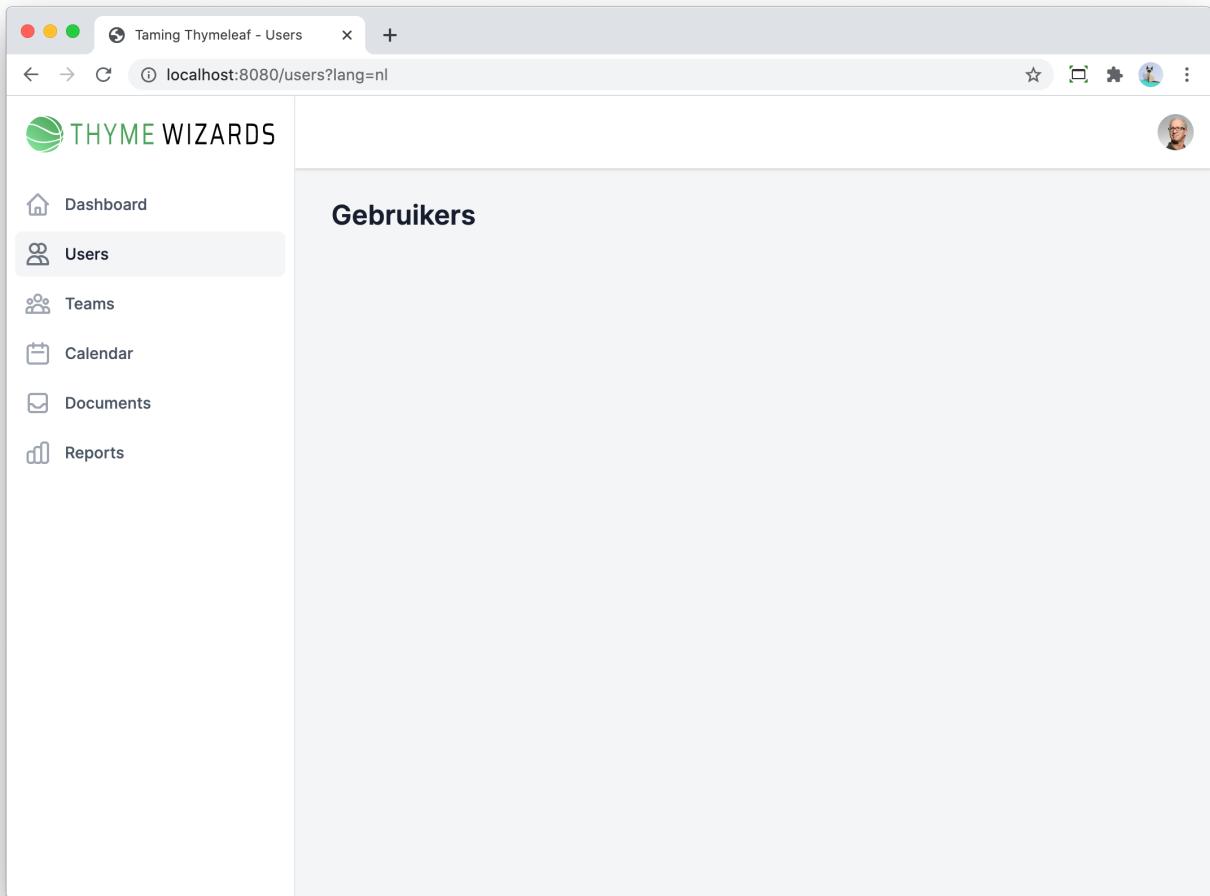


Figure 24. Language selection via `Lang` query parameter

If you now remove the query parameter and access <http://localhost:8080/users>, you will notice that the language remains Dutch. This is due to the cookie that has been stored. You can check this in the Developer Tools:

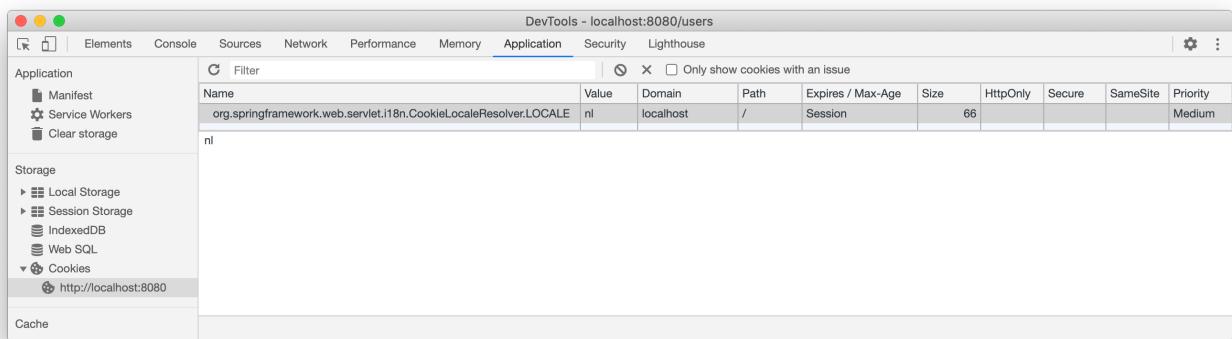


Figure 25. The locale cookie in Chrome Dev Tools

Remove the cookie or add `?Lang=en` to the URL to go back to the English translations.

## 8.3. Menu items translations

The menu items are currently not translated:

```
<nav class="flex-1 px-2 bg-white">
    <a th:replace="fragments/sidebar-buttons :: desktop-button(link='#{menu.dashboard}', title='Dashboard', menuItem='dashboard', icon='dashboard')"></a>
    <a th:replace="fragments/sidebar-buttons :: desktop-button(link=@{/users}, title='Users', menuItem='users', icon='users')"></a>
    <a th:replace="fragments/sidebar-buttons :: desktop-button(link=@{/teams}, title='Teams', menuItem='teams', icon='user-group')"></a>
    <a th:replace="fragments/sidebar-buttons :: desktop-button(link='#{menu.calendar}', title='Calendar', menuItem='calendar', icon='calendar')"></a>
    <a th:replace="fragments/sidebar-buttons :: desktop-button(link='#{menu.documents}', title='Documents', menuItem='documents', icon='documents')"></a>
    <a th:replace="fragments/sidebar-buttons :: desktop-button(link='#{menu.reports}', title='Reports', menuItem='reports', icon='reports')"></a>
</nav>
```

To translated them, add translations for each item to `messages.properties`:

```
users.title=Users
menu.dashboard=Dashboard
menu.users=Users
menu.teams=Teams
menu.calendar=Calendar
menu.documents=Documents
menu.reports=Reports
```

and `messages_nl.properties`:

```
users.title=Gebruikers
menu.dashboard=Dashboard
menu.users=Gebruikers
menu.teams=Teams
menu.calendar=Kalender
menu.documents=Documenten
menu.reports=Rapporten
```

Next, change `sidebar-menu.html` to use the translations:

```
<nav class="flex-1 px-2 bg-white">
    <a th:replace="fragments/sidebar-buttons :: desktop-button(link='#{menu.dashboard}', title='#{menu.dashboard}', menuItem='dashboard', icon='dashboard')"></a>
```

```

<a th:replace="fragments/sidebar-buttons :: desktop-
button(link=@{/users}, title=#${menu.users}, menuItem='users',
icon='users')"></a>
<a th:replace="fragments/sidebar-buttons :: desktop-
button(link=@{/teams}, title=#${menu.teams}, menuItem='teams',
icon='user-group')"></a>
<a th:replace="fragments/sidebar-buttons :: desktop-button(link='#',
title=#${menu.calendar}, menuItem='calendar', icon='calendar')"></a>
<a th:replace="fragments/sidebar-buttons :: desktop-button(link='#',
title=#${menu.documents}, menuItem='documents', icon='documents')"></a>
<a th:replace="fragments/sidebar-buttons :: desktop-button(link='#',
title=#${menu.reports}, menuItem='reports', icon='reports')"></a>
</nav>
```

As you can see, we can just use the translations as values for fragment arguments.

Adding or changing translations is not part of the auto-reload functionality, so you will need to restart the Spring Boot application when changing those.

If you forget to restart, or there is a key that is not in the translations files, then Thymeleaf will output the key surrounded with ??:

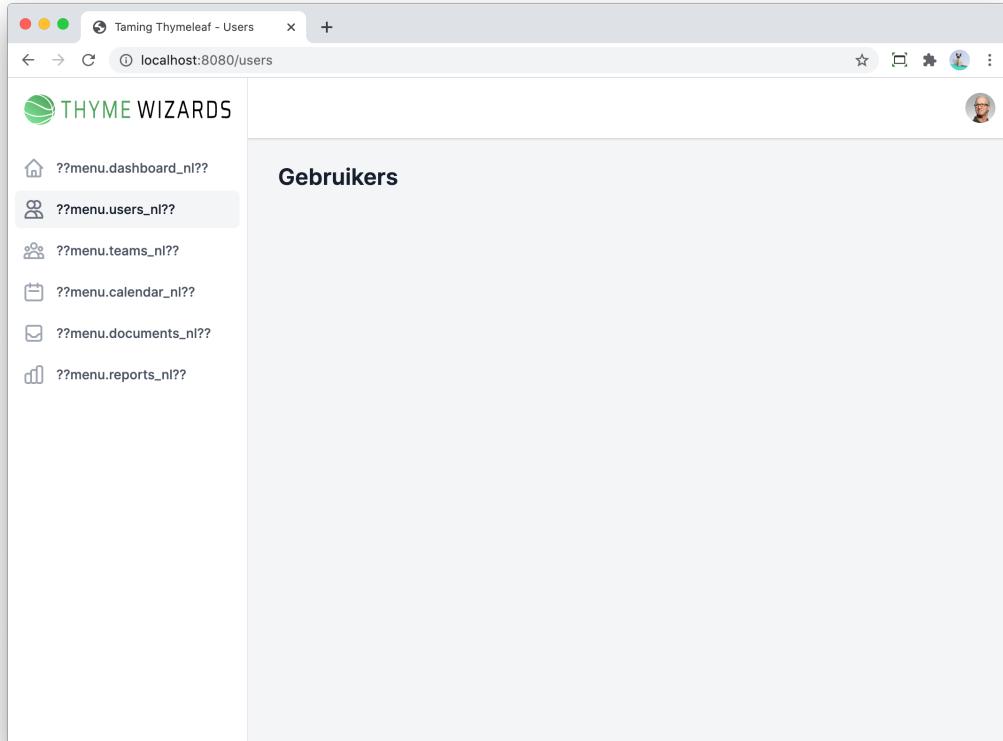


Figure 26. Thymeleaf unable to resolve translations

After these changes, the menu is properly translated:

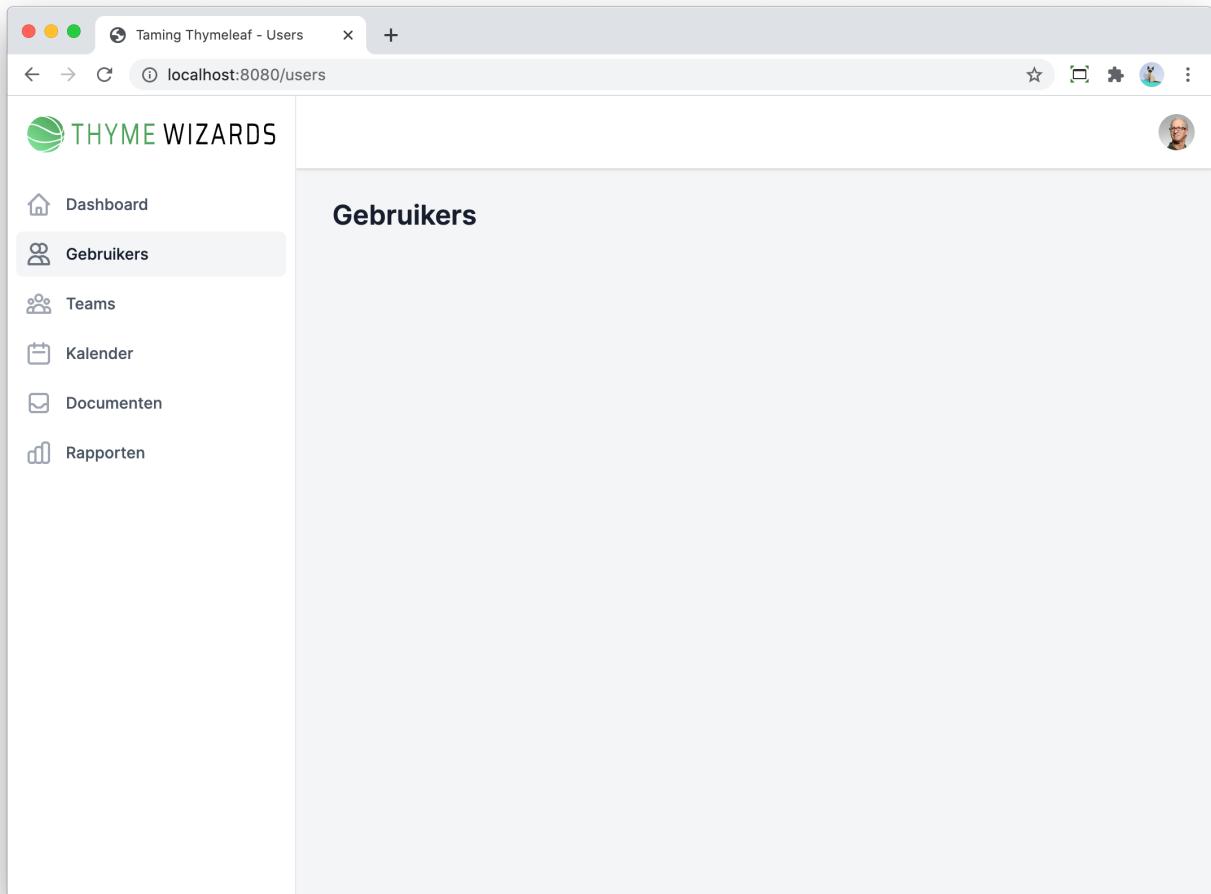


Figure 27. Menu translated in Dutch

## 8.4. Summary

In this chapter, you learned:

- How to add translations for different languages to your application.
- How to use the translations when rendering Thymeleaf templates.
- How to switch the used language based on query parameters or cookie values.

# Chapter 9. Database connection

## 9.1. Spring Data JPA

Our application does not *do* much at the moment. In an actual application, you will most likely do some database access to store and retrieve information. Thymeleaf and Spring MVC do not care at all what persistence technology you want to use. So feel free to use plain SQL (using [JdbcTemplate](#) or [JOOQ](#)), Java Persistence API, a NoSQL data store, or whatever else fits your use case in your projects.

For demonstration purposes, and because a lot of projects use it, we will implement the 4 CRUD (Create, Read, Update and Delete) actions using [Spring Data JPA](#) with [Hibernate](#) as the persistence provider.



[JPA](#) is the specification of a Java Object-Relational Mapper, while Hibernate is one of the possible implementations of that specification. If you are still confused about the difference between JPA and Hibernate, have a look at [this excellent stackoverflow answer](#) for more details.

## 9.2. PostgreSQL database

The database we will use is [PostgreSQL](#), but you can use any relational database that Hibernate supports (MySQL, Oracle, DB2, MS SQL Server, ...)

To quickly spin up a PostgreSQL database, we can use [Docker](#). Be sure to [install Docker](#) if you haven't before.

Create a [docker-compose.yaml](#) file in the root of the project:

*docker-compose.yaml*

```
version: '3'
services:
  db:
    image: 'postgres:12'
    ports:
      - "5432:5432"
    environment:
      POSTGRES_DB: ${POSTGRES_DATABASE}
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
```

Also create a [.env](#) file to specify the database, user and password to use:

```
POSTGRES_DATABASE=tamingthymeleafdb
POSTGRES_USER=postgres
POSTGRES_PASSWORD=FILL_IN
```



Add `.env` to your `.gitignore` file to avoid that you commit it by accident. In my projects, I do commit an `.env.example` file so other developers can create their own `.env` file easily.

Now start the database using:

```
docker-compose up -d
```

If you are using IntelliJ, you can also use the green arrows in the gutter when the `docker-compose.yaml` file is open in the editor:

```
1  version: '3'
2  docker-compose up
3  db:
4    image: 'postgres:12'
5    ports:
6      - "5432:5432"
7    environment:
8      POSTGRES_DB: ${POSTGRES_DATABASE}
9      POSTGRES_USER: ${POSTGRES_USER}
10     POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
```

Figure 28. IntelliJ allows to run docker-compose from the gutter

## 9.3. Getting started with Spring Data JPA

### 9.3.1. Add Spring Data JPA to the project

To add Spring Data JPA to the project, you need to add the following dependencies to the Maven `pom.xml`:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId> ①
</dependency>
```

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId> ②
</dependency>
<dependency>
    <groupId>io.github.wimdeblauwe</groupId>
    <artifactId>jpearl-core</artifactId> ③
    <version>${jpearl.version}</version>
</dependency>

```

① `spring-boot-starter-data-jpa` includes the JPA specification and Hibernate.

② `spring-boot-starter-validation` includes **Hibernate Validator** which allows to express and validate application constraints.

③ `jpearl` is a library with some utility classes I like to use when working with Spring Data JPA. See <https://github.com/wimdeblauwe/jpearl> for more information.

Further, add this to the `project > build > pluginManagement > plugins` section of the `pom.xml`:

```

<plugin>
    <groupId>io.github.wimdeblauwe</groupId>
    <artifactId>jpearl-maven-plugin</artifactId>
    <version>${jpearl.version}</version>
    <configuration>
        <basePackage>
com.tamingthymeleaf.application</basePackage>
        </configuration>
    </plugin>

```

The *JPA Early Primary Key Library* was created for 2 main reasons:

- It helps to implement *early primary key generation* whereby the primary key of the entities are generated before storing the object to the database.
  - By doing so, we avoid that there are objects that don't have a primary key yet until they are persisted to the database.
  - It makes implementing `equals()` and `hashCode` simpler since we know the primary key is passed at construction time and thus always present. If you are not using early primary key generation, be sure to follow the advice of Vlad Mihalcea at [The best way to implement equals, hashCode, and toString with JPA and Hibernate](#).
- It helps to use dedicated primary key classes which has the following advantages:
  - It more clearly expresses the intent. If a variable is of type `UserId`, it is clear what you are talking about, as opposed to a simple `Long` or `UUID`.
  - It is impossible to assign a value that is a `UserId` to an `OrderId` or a `BookId`. This reduces the chance of putting a wrong ID somewhere.
  - If you want to change from `UUID` to `long` or vice versa for the primary key, you will be able to do so with minimal changes to the application code.



These ideas are something I picked up from reading [Implementing Domain-Driven Design](#) by Vaughn Vernon.

### 9.3.2. User Entity

To store something in the database using Spring Data JPA, we need to define an entity. We will start with creating a [User](#) entity



If you are not familiar with the term *entity*, you should think of it as anything in your application that you want to identify and track over the lifetime of the thing. This can be users of an application, teams in our example application, orders in an e-commerce application, ...

This is in contrast with *value objects*. Those represent things that have no own identity. For example, an object that represents a distance, or an amount of money, ...

Using the JPearl Maven Plugin, we can generate the basic structure of our entity and the Spring Data JPA Repository.

Run:

```
mvn jpearl:generate -Dentity=User
```



You need to add the following to your [~/.m2/settings.xml](#) for the command to work:

```
<settings>
  <pluginGroups>
    <pluginGroup>io.github.wimdeblauwe</pluginGroup>
  </pluginGroups>
</settings>
```

The command should output something like this:

```
[INFO] --- jpearl-maven-plugin:0.3.0:generate (default-cli) @ taming-thymeleaf-application ---
[INFO] Generating entity: com.tamingthymeleaf.application.user.User
[INFO] Generating entity id: com.tamingthymeleaf.application.user.UserId
(using type: UUID)
[INFO] Generating entity repository:
com.tamingthymeleaf.application.user.UserRepository
[INFO] Generating entity repository custom:
com.tamingthymeleaf.application.user.UserRepositoryCustom
[INFO] Generating entity repository impl:
com.tamingthymeleaf.application.user.UserRepositoryImpl
```

```
[INFO] Generating entity repository test:
com.tamingthymeleaf.application.user.UserRepositoryTest
[INFO]
-----
[INFO] BUILD SUCCESS
[INFO]
-----
[INFO] Total time:  0.533 s
[INFO] Finished at: 2020-07-29T15:34:50+02:00
[INFO]
```

Let's take a look at the generated classes one by one.

This is the `User` entity class:

```
package com.tamingthymeleaf.application.user;

import io.github.wimdeblauwe.jpearl.AbstractEntity;

import javax.persistence.Entity;

@Entity ①
public class User extends AbstractEntity<UserId> { ②

    /**
     * Default constructor for JPA
     */
    protected User() { ③
    }

    public User(UserId id) { ④
        super(id);
    }
}
```

- ① Each *entity* must be annotated with the JPA defined `@Entity` annotation so our persistence library (Hibernate) knows that we want to persist these kind of objects.
- ② We extend from `AbstractEntity` which is a `jpearl` base class that defines that an entity must have a (unique) identifier. We are not using `Long` or `UUID` directly here, but a *value object* named `UserId`.
- ③ Hibernate requires a default constructor. We make it `protected` since our application code should never use that constructor directly.
- ④ The "normal" constructor requires an instance of the identifier.

The `UserId` class represents the primary key value object:

```
package com.tamingthymeleaf.application.user;

import io.github.wimdeblauwe.jpearl.AbstractEntityId;

import java.util.UUID;

public class UserId extends AbstractEntityId<UUID> { ①

    /**
     * Default constructor for JPA
     */
    protected UserId() { ②
    }

    public UserId(UUID id) { ③
        super(id);
    }
}
```

- ① The class extends from `AbstractEntityId` which is the base class for the identifier value objects. We use generics to specify that the underlying identifier is a `UUID`. If preferred, a `Long` could also be used.
- ② Hibernate requires a default constructor. We make it `protected` since our application code should never use that constructor directly.
- ③ The "normal" constructor requires an instance of the underlying identifier.

The reason for using a value object for the primary key is that it makes method signatures a lot clearer. It avoids mistakes where the primary key of one entity type is mistakenly used where the primary key of another entity type is required.

Suppose you have a method to add a user to a team. When using `Long`, you would have:



```
void addMemberToTeam(Long teamId, Long userId);
```

In contrast, using value objects, you have:

```
void addMemberToTeam(TeamId teamId, UserId userId);
```

It becomes impossible to mistakenly using the id of a user for the team and vice versa.

### 9.3.3. User repository

Next to `User` and `UserId`, the Maven plugin also generated:

- `UserRepository`
- `UserRepositoryCustom`
- `UserRepositoryImpl`

Looking at `UserRepository`:

```
package com.tamingthymeleaf.application.user;

import org.springframework.data.repository.CrudRepository;
import org.springframework.transaction.annotation.Transactional;

@Transactional(readOnly = true)
public interface UserRepository extends CrudRepository<User, UserId>,
UserRepositoryCustom {
}
```

This is just an `interface` that extends from the Spring Data JPA interface `CrudRepository` using our `User` and `UserId` as generics arguments.

Database transactions are important to ensure the database remains in a consistent state. Spring uses the `@Transactional` annotation to automatically apply transactions to methods (or all methods of a class when the annotation is applied on the class level).



For Spring Data Repositories, it is recommended to apply `@Transactional(readOnly=true)` on the class level, since most methods will be reading data. If a method is added that wants to write to the database, annotate that single method with `@Transactional(readOnly=false)` by default, so you can just leave it out).

If you want to let the database generate primary keys upon saving the entity, you only need this interface. However, we want to repository to generate unique id's. For this purpose, we need a `UserRepositoryCustom` interface:

```
package com.tamingthymeleaf.application.user;

public interface UserRepositoryCustom {
    UserId nextId();
}
```

That interface is implemented in `UserRepositoryImpl`:

```

package com.tamingthymeleaf.application.user;

import io.github.wimdeblauwe.jpearl.UniqueIdGenerator;

import java.util.UUID;

public class UserRepositoryImpl implements UserRepositoryCustom {
    private final UniqueIdGenerator<UUID> generator;

    public UserRepositoryImpl(UniqueIdGenerator<UUID> generator) { ①
        this.generator = generator;
    }

    @Override
    public UserId nextId() {
        return new UserId(generator.getNextUniqueId()); ②
    }
}

```

① Inject a `UniqueIdGenerator<UUID>`. This object is a Spring bean that will be responsible for generating unique UUID objects. JPearl has the `InMemoryUniqueIdGenerator` class that can do this for `UUIDs`. If you want to use `Long` objects instead, you will need to write your own implementation.

② Use the `UniqueIdGenerator` to get a new unique id and create a `UserId` instance.

At runtime, Spring Data JPA will combine their implementation of the `CrudRepository` with our custom Java code from `UserRepositoryImpl`. So if we inject the `UserRepository` interface into another object, that object can use the methods from the `CrudRepository` and the `UserRepositoryCustom` interfaces combined.

Update `TamingThymeleafApplicationConfiguration` to expose the `InMemoryUniqueIdGenerator` as a bean in the application:

`com.tamingthymeleaf.application.TamingThymeleafApplicationConfiguration`

```

@Bean
public UniqueIdGenerator<UUID> uniqueIdGenerator() {
    return new InMemoryUniqueIdGenerator();
}

```

## Package structure

The code in this book uses *package by feature*. This way of structuring packages creates a separate package for each feature in the application. So `user`, `team`, `game`, ... Inside each package, we will find the domain objects and services (For the `user` package this would be `User`, `UserService`, `UserRepository`, ... ).

The things that are not directly related to the domain like web controllers are put in a [web](#) sub-package.

Next to those *feature packages*, there is also 1 [infrastructure](#) package that contains all code related to infrastructure concerns like security, validation, serialization, ...

### 9.3.4. User Repository Test

The JPearl Maven Plugin also generates an integration test using [JUnit 5, AssertJ](#) and the [Spring Testing](#) support.

Since we are writing very little actual code, but rely on JPA annotations and Spring Data JPA, it is better to write an integration test to ensure everything is working well. A unit test in the strict sense would have little benefit here.

This is the code of the generated [UserRepositoryTest](#):

```
package com.tamingthymeleaf.application.user;

import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import java.util.UUID;

import static org.assertj.core.api.Assertions.assertThat;

@DataJpaTest ①
class UserRepositoryTest {
    private final UserRepository repository;
    private final JdbcTemplate jdbcTemplate;
    @PersistenceContext ②
    private EntityManager entityManager;

    @Autowired
    UserRepositoryTest(UserRepository repository,
                      JdbcTemplate jdbcTemplate) { ③
        this.repository = repository;
        this.jdbcTemplate = jdbcTemplate;
    }

    @BeforeEach
}
```

```

void validatePreconditions() { ④
    assertThat(repository.count()).isZero();
}

@Test
void testSaveUser() { ⑤
    UserId id = repository.nextId();
    repository.save(new User(id)); ⑥

    entityManager.flush(); ⑦

    UUID idInDb = jdbcTemplate.queryForObject("SELECT id FROM user",
UUID.class); ⑧
    assertThat(idInDb).isEqualTo(id.getId()); ⑨
}
}

```

- ① `@DataJpaTest` instructs the Spring testing support that this test only needs "things" related to database and persistence. Services and web controllers will not be started to speed up the tests.
- ② We inject the `EntityManager` via the `@PersistenceContext` annotation so we can flush the JPA statements and validate the actual database tables.
- ③ We inject the `UserRepository` since that is the object we want to test, and the `JdbcTemplate` which will help us validate the contents of the database.
- ④ Before each test starts, we validate that the database is empty. This ensures we start each test from a valid state.
- ⑤ `testSaveUser` is our actual test method.
- ⑥ We use the `save()` method from `UserRepository` to store an instance of the user.
- ⑦ We call `flush()` to have Hibernate write all changes to the database.
- ⑧ By using `jdbcTemplate.queryForObject`, we can validate if the generated unique id was persisted in the `user` database table.
- ⑨ Assert that the id from the database matches with the generated id.



We inject the `EntityManager` using `@PersistenceContext` here. We could have used `@Autowired` as well, but it is better to use `@PersistenceContext`. See [@Autowired vs @PersistenceContext for EntityManager bean](#) for details.

Before we can run the test, we need database tables. We can have Hibernate do this automatically, but that is not a good solution for an actual production-grade application.

It is better to use either [Flyway](#) or [Liquibase](#). We will go with Flyway in our application.

Add the Flyway dependency in the `pom.xml`:

```

<dependency>
    <groupId>org.flywaydb</groupId>

```

```
<artifactId>flyway-core</artifactId>
</dependency>
```

Now create [src/main/resources/db/migration/V1.0\\_\\_init.sql](#):

```
CREATE TABLE tt_user
(
    id UUID NOT NULL,
    PRIMARY KEY (id)
);
```

We are using `tt_user` here as table name instead of `user` as PostgreSQL does not allow it (unless you quote the table name always).

As a result of this, we need to tell Hibernate to use that table name via the `@Table` annotation:



```
@Entity
@Table(name = "tt_user")
public class User extends AbstractEntity<UserId> {
```

Flyway will automatically run the `V1.0__init.sql` script at startup of the application or a `@DataJpaTest`. It will also make note of this in a special table so it won't run the script again on next starts of the application.

The final step before we can run our database test is having a database to run against. We could use an in-memory database like H2 for that, however, it is better to test against an actual PostgreSQL database. This used to be a big hassle to set up, but thanks to Docker and `Testcontainers`, this is no longer the case.

Add the needed dependencies to the `pom.xml`:

```
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>testcontainers</artifactId>
    <version>${testcontainers.version}</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>postgresql</artifactId>
    <version>${testcontainers.version}</version>
    <scope>test</scope>
</dependency>
```

We also need to add the PostgreSQL driver dependency:

```
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
</dependency>
```

We can now configure `UserRepositoryTest` to use an actual PostgreSQL database started via Testcontainers:

```
package com.tamingthymeleaf.application.user;

import io.github.wimdeblauwe.jpearl.InMemoryUniqueIdGenerator;
import io.github.wimdeblauwe.jpearl.UniqueIdGenerator;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDatabase;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.context.annotation.Bean;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.test.context.ActiveProfiles;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import java.util.UUID;

import static org.assertj.core.api.Assertions.assertThat;

@DataJpaTest
@ActiveProfiles("data-jpa-test") ①
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace
.NONE) ②
class UserRepositoryTest {
    private final UserRepository repository;
    private final JdbcTemplate jdbcTemplate;
    @PersistenceContext
    private EntityManager entityManager;

    @Autowired
```

```

UserRepositoryTest(UserRepository repository,
                  JdbcTemplate jdbcTemplate) {
    this.repository = repository;
    this.jdbcTemplate = jdbcTemplate;
}

@BeforeEach
void validatePreconditions() {
    assertThat(repository.count()).isZero();
}

@Test
void testSaveUser() {
    UserId id = repository.nextId();
    repository.save(new User(id));

    entityManager.flush();

    UUID idInDb = jdbcTemplate.queryForObject("SELECT id FROM
tt_user", UUID.class); ③
    assertThat(idInDb).isEqualTo(id.getId());
}

@TestConfiguration ④
static class TestConfig {
    @Bean
    public UniqueIdGenerator<UUID> uniqueIdGenerator() { ⑤
        return new InMemoryUniqueIdGenerator();
    }
}
}

```

① The `ActiveProfiles` annotation allows to activate a certain profile when the test runs. By specifying `data-jpa-test`, we can set properties in an `application-data-jpa-test.properties` file and they will be used when the test runs.

② Spring Test will by default try to setup an in-memory database. We need to opt-out of that by using `@AutoConfigureTestDatabase` since we are using Testcontainers.

③ Use the `tt_user` table.

④ An inner class that is annotated with `@TestConfiguration` will be added to the Spring context that is started by the Spring Testing framework. This allows us to define the `UniqueIdGenerator` bean that our repository needs.

We are using the `JDBC support` of Testcontainers by using a special JDBC URL which will instruct Testcontainers to start a Docker image with PostgreSQL and make it available to our test. By activating the `data-jpa-test` profile, the test will read the `application-data-jpa-test.properties` file, that has the needed properties to make it all work:

*src/test/resources/application-data-jpa-test.properties*

```
spring.datasource.url=jdbc:tc:postgresql:12://tamingthymeleafdb?TC_TMPF
S=/testtmpfs:rw ①
spring.datasource.driver-class-name
=org.testcontainers.jdbc.ContainerDatabaseDriver ②
spring.datasource.username=user ③
spring.datasource.password=password ④
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect ⑤
spring.jpa.hibernate.ddl-auto=validate ⑥

logging.level.org.hibernate.SQL=DEBUG ⑦
spring.jpa.properties.hibernate.show_sql=false ⑧
```

- ① Set the JDBC URL for Testcontainers to start PostgreSQL 12.
- ② Use the Testcontainers JDBC driver.
- ③ Testcontainers uses `user` as the username to access the database.
- ④ Testcontainers uses `password` as the password to access the database.
- ⑤ Instruct Hibernate that PostgreSQL dialect is used.
- ⑥ Have Hibernate validate if the database tables match with the entity mappings at startup.
- ⑦ Enable the `org.hibernate.SQL` logging level so the SQL statements are printed using the logging framework.
- ⑧ Disable the SQL logging of Hibernate itself because that logs to standard out and not the logging framework.

With all that in place, we can finally run our [UserRepositoryTest](#).

The abbreviated output should look similar to this:

```
2020-07-30 16:08:47.350 INFO 13632 --- [           main]
c.t.application.user.UserRepositoryTest : Starting UserRepositoryTest
on Wims-MacBook-Pro.local with PID 13632
2020-07-30 16:08:47.350 INFO 13632 --- [           main]
c.t.application.user.UserRepositoryTest : The following profiles are
active: data-jpa-test
...
2020-07-30 16:08:49.285 INFO 13632 --- [           main] 🐳
[postgres:12]                               : Creating container for image:
postgres:12
...
2020-07-30 16:08:50.675 INFO 13632 --- [           main]
o.f.c.internal.database.DatabaseFactory : Database:
jdbc:postgresql://localhost:32793/test (PostgreSQL 12.1)
2020-07-30 16:08:50.716 INFO 13632 --- [           main]
o.f.core.internal.command.DbValidate    : Successfully validated 1
```

```

migration (execution time 00:00.015s)
2020-07-30 16:08:50.734 INFO 13632 --- [           main]
o.f.c.i.s.JdbcTableSchemaHistory       : Creating Schema History table
"public"."flyway_schema_history" ...
2020-07-30 16:08:50.763 INFO 13632 --- [           main]
o.f.core.internal.command.DbMigrate    : Current version of schema
"public": <> Empty Schema >>
2020-07-30 16:08:50.772 INFO 13632 --- [           main]
o.f.core.internal.command.DbMigrate    : Migrating schema "public" to
version 1.0 - init
2020-07-30 16:08:50.794 INFO 13632 --- [           main]
o.f.core.internal.command.DbMigrate    : Successfully applied 1
migration to schema "public" (execution time 00:00.038s)
...
2020-07-30 16:08:51.891 INFO 13632 --- [           main]
c.t.application.user.UserRepositoryTest : Started UserRepositoryTest in
4.83 seconds (JVM running for 5.677)
...
2020-07-30 16:08:51.972 INFO 13632 --- [           main]
o.s.t.c.transaction.TransactionContext : Began transaction (1) for
test ...
2020-07-30 16:08:52.244 DEBUG 13632 --- [           main]
org.hibernate.SQL                      : select count(*) as col_0_0_
from tt_user user0_
2020-07-30 16:08:52.299 DEBUG 13632 --- [           main]
org.hibernate.SQL                      : select user0_.id as id1_0_0_
from tt_user user0_ where user0_.id=?
2020-07-30 16:08:52.316 DEBUG 13632 --- [           main]
org.hibernate.SQL                      : insert into tt_user (id)
values (?)
2020-07-30 16:08:52.335 INFO 13632 --- [           main]
o.s.t.c.transaction.TransactionContext : Rolled back transaction for
test ...
...

```

So the flow is:

1. Test starts.
2. A Docker container is created using the `postgres:12` image.
3. Flyway runs the database migrations against the PostgreSQL database.
4. Spring starts a transaction.
5. Our actual test code runs.
6. Spring rolls back the transaction to ensure the database is back in the original state.

### 9.3.5. Adding properties to User

So far, our `User` only has a surrogate primary key (the `id`). Let's add some more fields to make things interesting:

```
package com.tamingthymeleaf.application.user;

import io.github.wimdeblauwe.jpearl.AbstractEntity;

import javax.persistence.Entity;
import javax.persistence.EnumType;
import javax.persistence.Enumerated;
import javax.persistence.Table;
import javax.validation.constraints.NotNull;
import java.time.LocalDate;

@Entity
@Table(name = "tt_user")
public class User extends AbstractEntity<UserId> {

    @NotNull
    private UserName userName; ①
    @NotNull
    @Enumerated(EnumType.STRING)
    private Gender gender; ②
    @NotNull
    private LocalDate birthday; ③
    @NotNull
    private Email email; ④
    @NotNull
    private PhoneNumber phoneNumber; ⑤

    protected User() {
    }

    public User(UserId id,
               UserName userName,
               Gender gender,
               LocalDate birthday,
               Email email,
               PhoneNumber phoneNumber) {
        super(id);
        this.userName = userName;
        this.gender = gender;
        this.birthday = birthday;
```

```

        this.email = email;
        this.phoneNumber = phoneNumber;
    }

    public UserName getUserName() {
        return userName;
    }

    public Gender getGender() {
        return gender;
    }

    public LocalDate getBirthday() {
        return birthday;
    }

    public Email getEmail() {
        return email;
    }

    public PhoneNumber getPhoneNumber() {
        return phoneNumber;
    }
}

```

- ① **UserName** is a value object that contains the **firstName** and **lastName** for a user.
- ② **Gender** is an **enum** for the possible genders of a user in our application.
- ③ The **birthday** field is using **LocalDate** to store the day a user was born.
- ④ **Email** is a value object that represents an email address.
- ⑤ **PhoneNumber** is a value object that represents a phone number.

Some things to note:

- All fields are required, so we have annotated them with **javax.validation.constraints.NotNull**
- By default, an **enum** is serialized to the database using its ordinal (an **int**). By adding **@Enumerated(EnumType.STRING)**, Hibernate will write the **name** of the enum to the database.
- For now, only getters have been added. This is a pattern I like to use where I will only add setters when there is an actual use case for changing something.
- **UserName** is annotated with **@Embeddable** so that the **firstName** and **lastName** fields are inlined into the **tt\_user** database table.
- The value objects **Email** and **PhoneNumber** have a JPA **AttributeConverter** class that knows how to convert from the value object to a String for the database and back. Example:

```

package com.tamingthymeleaf.application.user;

import javax.persistence.AttributeConverter;
import javax.persistence.Converter;

@Converter(autoApply = true) ①
public class PhoneNumberAttributeConverter implements
AttributeConverter<PhoneNumber, String> {
    @Override
    public String convertToDatabaseColumn(PhoneNumber attribute) {
        return attribute.asString();
    }

    @Override
    public PhoneNumber convertToEntityAttribute(String dbData) {
        return new PhoneNumber(dbData);
    }
}

```

① Spring Boot will automatically apply the converter. The `autoApply` indicates that this converter should be used for all `PhoneNumber` typed fields across the application.

- Hibernate can automatically map from a `LocalDate` to the `DATE` column type.

See the [sources on GitHub](#) for the full details of `UserName`, `Email` and `PhoneNumber` value objects, as well as `EmailAttributeConverter` and `PhoneNumberAttributeConverter`.

Note that we also added Guava as a dependency (for the `toString()` implementation):



```

<dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>${guava.version}</version>
</dependency>

```

To ensure all is ok, we update our `UserRepositoryTest`:

`com.tamingthymeleaf.application.user.UserRepositoryTest`

```

@Test
void testSaveUser() {
    UserId id = repository.nextId();
    repository.save(new User(id,

```

```

        new UserName("Tommy", "Walton"),
        Gender.MALE,
        LocalDate.of(2001, Month.FEBRUARY, 17),
        new Email("tommy.walton@gmail.com"),
        new PhoneNumber("202 555 0192")));

entityManager.flush();

assertThat(jdbcTemplate.queryForObject("SELECT id FROM tt_user",
UUID.class)).isEqualTo(id.getId());
assertThat(jdbcTemplate.queryForObject("SELECT first_name FROM
tt_user", String.class)).isEqualTo("Tommy");
assertThat(jdbcTemplate.queryForObject("SELECT last_name FROM
tt_user", String.class)).isEqualTo("Walton");
assertThat(jdbcTemplate.queryForObject("SELECT gender FROM
tt_user", Gender.class)).isEqualTo(Gender.MALE);
assertThat(jdbcTemplate.queryForObject("SELECT birthday FROM
tt_user", LocalDate.class)).isEqualTo("2001-02-17");
assertThat(jdbcTemplate.queryForObject("SELECT email FROM
tt_user", String.class)).isEqualTo("tommy.walton@gmail.com");
assertThat(jdbcTemplate.queryForObject("SELECT phone_number FROM
tt_user", String.class)).isEqualTo("202 555 0192");
}

```

Wait! If we would run this now, it would fail as our Flyway migration script is not yet updated. Update it now to this:

*src/main/resources/db/migration/V1.0\_init.sql*

```

CREATE TABLE tt_user
(
    id          UUID      NOT NULL,
    first_name  VARCHAR   NOT NULL,
    last_name   VARCHAR   NOT NULL,
    gender      VARCHAR   NOT NULL,
    birthday    DATE      NOT NULL,
    email       VARCHAR   NOT NULL,
    phone_number VARCHAR  NOT NULL,
    PRIMARY KEY (id)
);

```

After that our [UserRepositoryTest](#) should be all green.

The [TamingThymeleafApplicationTests](#) will still fail. We need to also start a Testcontainers based database for it to work.

Add the `@ActiveProfiles` and `@AutoConfigureTestDatabase` annotations to the test class like this:

`com.tamingthymeleaf.application.user.TamingThymeleafApplicationTests.java`

```
@SpringBootTest
@ActiveProfiles("spring-boot-test")
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace
.NONE)
class TamingThymeleafApplicationTests {
```

Add `application-spring-boot-test.properties` properties file to `src/test/resources`:

`src/test/resources/application-spring-boot-test.properties`

```
spring.datasource.url=jdbc:tc:postgresql:12://tamingthymeleafdb?TC_TMPF
S=/testtmpfs:rw
spring.datasource.driver-class-name
=org.testcontainers.jdbc.ContainerDatabaseDriver
spring.datasource.username=user
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.hibernate.ddl-auto=validate
logging.level.org.hibernate.SQL=DEBUG
spring.jpa.properties.hibernate.show_sql=false
```

All tests should be ok now.

The application will not start at this point if you try to start it. It will fail with:

```
*****
APPLICATION FAILED TO START
*****
```

Description:

Failed to configure a DataSource: 'url' attribute is not specified and no embedded datasource could be configured.

Reason: Failed to determine a suitable driver class



Do not worry. We will fix this in the next chapter when we will display data from the database in our application.

## 9.4. Summary

In this chapter, you learned:

- How to use Spring Data JPA.
- How to create entities for mapping a Java object to a database table.
- How to create a repository for interaction with the database.

# Chapter 10. Displaying data

Let's get back to the UI part of the application and show a table of existing users.

## 10.1. Generate random users

The first thing we need is a few users in our database for testing. We could create a database script to do that, but I find it easier to write a simple Java class that can create stuff in a loop. Start by creating a `CommandLineRunner` implementation. Spring will run any such beans at startup of the application.

By only enabling the bean when the `init-db` profile is active, we can toggle if the database should be populated at startup or not.

This is the code for the `DatabaseInitializer`:

```
package com.tamingthymeleaf.application;

import com.github.javafaker.Faker;
import com.github.javafaker.Name;
import com.tamingthymeleaf.application.user.*;
import org.apache.commons.lang3.StringUtils;
import org.springframework.boot.CommandLineRunner;
import org.springframework.context.annotation.Profile;
import org.springframework.stereotype.Component;

import java.time.LocalDate;
import java.time.ZoneId;

@Component
@Profile("init-db") ①
public class DatabaseInitializer implements CommandLineRunner { ②
    private final Faker faker = new Faker(); ③
    private final UserService userService;

    public DatabaseInitializer(UserService userService) { ④
        this.userService = userService;
    }

    @Override
    public void run(String... args) {
        for (int i = 0; i < 20; i++) { ⑤
            CreateUserParameters parameters = newRandomUserParameters();
            userService.createUser(parameters);
        }
    }
}
```

```

private CreateUserParameters newRandomUserParameters() {
    Name name = faker.name();
    UserName userName = new UserName(name.firstName(), name.lastName());
    Gender gender = faker.bool().bool() ? Gender.MALE : Gender.FEMALE;
    LocalDate birthday = LocalDate.ofInstant(faker.date().birthday(10, 40).toInstant(), ZoneId.systemDefault());
    Email email = new Email(faker.internet().emailAddress(generateEmailLocalPart(userName)));
    PhoneNumber phoneNumber = new PhoneNumber(faker.phoneNumber().phoneNumber());
    return new CreateUserParameters(userName, gender, birthday, email, phoneNumber);
}

private String generateEmailLocalPart(UserName userName) {
    return String.format("%s.%s",
        StringUtils.remove(userName.getFirstName().toLowerCase(), ""),
        StringUtils.remove(userName.getLastName().toLowerCase(), ""));
}

```

- ① Only have this `@Component` active when the `init-db` profile is active.
- ② Implement `CommandLineRunner` interface so that Spring calls the `run()` method at startup.
- ③ Use `java-faker` to generate random names, birthdays, email addresses, ...

This requires this dependency in the `pom.xml`:

```

<dependency>
    <groupId>com.github.javafaker</groupId>
    <artifactId>javafaker</artifactId>
    <version>${javafaker.version}</version>
</dependency>

```

- ④ Our initializer will use the `UserService` interface to create and persist `User` objects.
- ⑤ Loop 20 times to generate 20 different users.

There are 2 new classes used here: `UserService` and `CreateUserParameters`.

`UserService` is in fact an interface that exposes methods related to the `User` entity and internally will depend on the `UserRepository` to persist the entities. The `UserController` will also need this

service as that will contain all the business logic, as to keep the controller as small as possible.

```
package com.tamingthymeleaf.application.user;

public interface UserService {
    User createUser(CreateUserParameters parameters);
}
```

The `UserService` currently only has a single method `createUser` with a single argument `CreateUserParameters`:

```
package com.tamingthymeleaf.application.user;

import java.time.LocalDate;

public class CreateUserParameters {
    private final UserName userName;
    private final Gender gender;
    private final LocalDate birthday;
    private final Email email;
    private final PhoneNumber phoneNumber;

    public CreateUserParameters(UserName userName,
                               Gender gender,
                               LocalDate birthday,
                               Email email,
                               PhoneNumber phoneNumber) {
        this.userName = userName;
        this.gender = gender;
        this.birthday = birthday;
        this.email = email;
        this.phoneNumber = phoneNumber;
    }

    public UserName getUserName() {
        return userName;
    }

    public Gender getGender() {
        return gender;
    }

    public LocalDate getBirthday() {
        return birthday;
    }
}
```

```

    }

    public Email getEmail() {
        return email;
    }

    public PhoneNumber getPhoneNumber() {
        return phoneNumber;
    }
}

```

Which is very similar to `User` itself currently without the `id` field.

## Parameters vs FormData objects

Throughout the book, there will be `...Parameters` classes and `...FormData` classes.

A `FormData` class is always used as a form backing object for a HTML form. It will typically be mutable as Spring MVC needs to bind the changes from the form on the object. It will *not* throw `NullPointerException` or `IllegalArgumentException` when there is invalid data, because the class exists to model invalid data entered by a user in a form. We need to have that invalid data when we re-render the page to show what is wrong and give the user a chance to fix their mistake.

It will also typically use `String` typing since that is easy to use for binding. It is part of the *web* layer within the feature package and should only be used by the controller.

A `Parameters` object is part of the *domain* layer. Such objects will be immutable and they will validate all constructor arguments, throwing `NullPointerException`, `IllegalArgumentException` or other exception types as needed.

An additional advantage of a `Parameters` object is that you can use it as an alternative to having each possible parameter in the method signature.

To make that more understandable, it is the difference between:

```

public interface UserService {
    User createUser(String firstName, String lastName, String
password, Gender gender, ...);
}

```

and:

```

public interface UserService {
    User createUser(CreateUserParameters parameters);
}

```

Since the *web* layer can depend on the *domain* layer, but not vice-versa, the `FormData` object will usually have a `toParameters()` method to convert from the form representation to the domain-level `Parameters` representation.

The implementation of the `UserService` interface is done in `UserServiceImpl`:

```
@Service ①
@Transactional ②
public class UserServiceImpl implements UserService {
    private final UserRepository repository;

    public UserServiceImpl(UserRepository repository) {
        this.repository = repository;
    }

    @Override
    public User createUser(CreateUserParameters parameters) {
        UserId userId = repository.nextId(); ③
        User user = new User(userId,
            parameters.getUserName(),
            parameters.getGender(),
            parameters.getBirthday(),
            parameters.getEmail(),
            parameters.getPhoneNumber()); ④
        return repository.save(user); ⑤
    }
}
```

- ① Add the `@Service` annotation so Spring automatically creates an instance in the context.
- ② Add the `@Transactional` annotation to ensure each method is wrapped in a transaction.
- ③ Ask the repository to generate a new unique id.
- ④ Create a `User` object with the id and the values from the `parameters` object.
- ⑤ Persist the `user` to the database and return it.

Before we now can start our application to generate the 20 random users, we need to tell it where to find the database. If you followed along, you should have a `PostgreSQL database` running in Docker.

Update `application-local.properties` to look like this:

```
spring.thymeleaf.cache=false
# Database setup
spring.datasource.url=jdbc:postgresql://localhost/tamingthymeleafdb
spring.datasource.driver-class-name=org.postgresql.Driver
spring.datasource.username=postgres
```

```
spring.datasource.password=PUT_YOUR_PWD_HERE
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.hibernate.ddl-auto=validate
```



Don't forget to set the correct password you used in your `.env` file for the `spring.datasource.password` property.



Put `application-local.properties` in your `.gitignore` file and commit an `application-local.properties.template` file with the sensitive information removed to avoid accidental commits of your password.

Now run the application with the `local,init-db` profiles.

Connect to the database using your favorite tool (I just use IntelliJ IDEA) and see that our 20 users are now present in the `tt_user` table:

The screenshot shows the IntelliJ IDEA Database tool interface. On the left, a table view displays the `tt_user` table with 20 rows of data. The columns are: id, first\_name, last\_name, gender, birthday, email, and phone\_number. The data includes various names like Alberta, Ned, Ethelene, Brendon, Leo, Brenton, Casimira, Leigh, Guy, Reuben, Michal, Raeann, Rigoberto, Clair, Sau, Jerald, Shannan, Rosario, Erin, and Matha, along with their respective gender, birthdates, emails, and phone numbers. On the right, the Database browser shows the connection to `postgres@localhost`, the schema `tamingthymeleafdb`, and the table `tt_user`.

	<code>id</code>	<code>first_name</code>	<code>last_name</code>	<code>gender</code>	<code>birthday</code>	<code>email</code>	<code>phone_number</code>
1	f4ee36b6-1b0a-4a40-b015-e3e2724bd076	Alberta	Jerde	MALE	2003-06-19	alberta.jerde@hotmail.com	923.070.8338
2	b8a608ef-f9f3-4d90-8b81-e1a1b0980e03	Ned	Murphy	FEMALE	1993-09-26	ned.murphy@gmail.com	917.601.6007
3	566f44ba-b773-4592-a2da-4ec711dedd8f	Ethelene	Haag	FEMALE	1981-11-26	ethelene.haag@gmail.com	1-952-852-6407 x6296
4	c72fb26c-c05e-4336-b284-4f4979ea5c2f	Brendon	Schamberger	MALE	1998-10-17	brendon.schamberger@hotmail.com	256-246-0457 x1829
5	7c1fe911-6e68-404d-a25b-2f591e1d1f5f	Lee	Legros	FEMALE	2009-08-23	leo.legros@yahoo.com	1-653-516-1324 x1533
6	7c6cec9f-5b04-4460-afe-e196ebfd340d	Brenton	Ledner	MALE	1989-04-20	brenton.ledner@yahoo.com	1-855-608-6301 x523
7	38705463-d88e-4d76-8462-e395ab01ad8e	Casimira	Dubuque	FEMALE	1986-09-11	casimira.dubuque@yahoo.com	1-988-862-3715 x5726
8	cff90472e-dbc1-4c96-9ab0-74f93c0cbf8a	Leigh	Monahan	MALE	1985-12-18	leigh.monahan@gmail.com	200.115.9286 x304
9	ce4de631-fa71-4f7a-9962-e1f81a1a95577	Guy	O'Connell	MALE	1981-12-31	guy.oconnell@gmail.com	(863) 826-1357 x6783
10	7e664094-208c-4665-97b8-a69429495eae	Reuben	Schinner	MALE	1990-02-13	reuben.schinner@yahoo.com	1-363-883-9354 x7297
11	5ce61d45-5783-4f3d-9672-55f2e61eb2b3	Michal	Rogahn	MALE	1988-08-11	michal.rogahn@gmail.com	1-195-392-4297
12	c8d289a7-8c39-4f7b-81fa-5c10845f946d	Raeann	Boehm	MALE	1994-02-18	raeann.boehm@gmail.com	1-148-262-6332 x79866
13	d7a60183-14dc-4aac-8dfe-c120a368e1fa	Rigoberto	Feehey	MALE	1995-06-09	rigoberto.feehey@hotmail.com	347.298.1487 x6800
14	9f19245b-18d6-40ca-96fe-f9317ae6eeSeed	Clair	Gislason	MALE	1994-10-04	clair.gislason@hotmail.com	367-591-8634
15	0bb7a55c-912a-4594-a216-823bf20ca936	Sau	Ferry	MALE	1995-04-15	sau.ferry@yahoo.com	598-888-7669 x487
16	f523218e-305f-425d-a699-ead02a79a6d6	Jerald	Welch	FEMALE	1994-07-26	jerald.welch@gmail.com	924-476-3729
17	e67fd43c-c542-45f7-98b4-c4e81416112b	Shannan	Gaylord	MALE	1996-03-18	shannan.gaylord@hotmail.com	987.884.0851
18	c76710ec-c974-4ddf-acdb-8a9312359dd0	Rosario	Barton	MALE	2005-06-24	rosario.barton@yahoo.com	716.676.1156
19	1d92f205-beef-401f-bd18-45427c6bc7ff	Erin	Schuster	FEMALE	1989-06-17	erin.schuster@hotmail.com	(541) 498-6468 x6158
20	34ce68e3-4eee-46c3-a369-976b35acd983	Matha	Bradtke	MALE	1980-11-20	matha.bradtke@hotmail.com	834.628.2617

Figure 29. The `tt_user` database table with 20 randomly generated users

Stop the application again and ensure you start with the `local` profile only the next time, or another 20 users will be added to the database.

## 10.2. Get users on the HTML page

Now we have some users in our database, it is time to display them in the UI.

We need to take 3 steps for this:

1. Get the users from the database.
2. Put the users in the model that Thymeleaf uses to render the HTML.
3. Generate table rows in the Thymeleaf template for each of the users.

To get the users from the database, we update `UserService` with a `getAllUsers` method:

```
@Override
public ImmutableSet<User> getAllUsers() {
    return ImmutableSet.copyOf(repository.findAll());
```

```
}
```

This uses the `ImmutableSet` class from Guava since the result of the method call should not be altered.

Next, we inject `UserService` into `UserController` and put the users in the model under the `users` key:

```
package com.tamingthymeleaf.application.user.web;

import com.tamingthymeleaf.application.user.UserService;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
@RequestMapping("/users")
public class UserController {

    private final UserService service;

    public UserController(UserService service) {
        this.service = service;
    }

    @GetMapping
    public String index(Model model) {
        model.addAttribute("users", service.getAllUsers()); ①
        return "users/list"; ②
    }
}
```

① Store the returned users under the `users` key in the model.

② Return the name of the Thymeleaf view to render. `users/list` means that the view at `src/main/resources/templates/users/list.html` will be used.

Finally, we update the `list.html` template to render each user:

```
<!DOCTYPE html>
<html
    xmlns:th="http://www.thymeleaf.org"
    xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
    layout:decorate="~{layout/layout}"
    th:with="activeMenuItem='users'">
```

```

<head>
    <title>Users</title>
</head>
<body>
<div layout:fragment="page-content">
    <div class="max-w-7xl mx-auto px-4 sm:px-6 md:px-8">
        <h1 class="text-2xl font-semibold text-gray-900"
            th:text="#{users.title}">Users</h1>
    </div>
    <div class="max-w-7xl mx-auto px-4 sm:px-6 md:px-8">
        <div class="py-4">
            <div class="flex flex-col">
                <div class="-my-2 py-2 overflow-x-auto sm:-mx-6 sm:px-6
lg:-mx-8 lg:px-8">
                    <div class="align-middle inline-block min-w-full
shadow overflow-hidden rounded-md sm:rounded-lg border-b border-gray-
200">
                        <table class="min-w-full">
                            <thead>
                                <tr>
                                    <th class="px-6 py-3 border-b border-
gray-200 bg-gray-50 text-left text-xs leading-4 font-medium text-gray-
500 uppercase tracking-wider">
                                        Name
                                    </th>
                                    <th class="px-6 py-3 border-b border-
gray-200 bg-gray-50 text-left text-xs leading-4 font-medium text-gray-
500 uppercase tracking-wider">
                                        Gender
                                    </th>
                                    <th class="px-6 py-3 border-b border-
gray-200 bg-gray-50 text-left text-xs leading-4 font-medium text-gray-
500 uppercase tracking-wider">
                                        Birthday
                                    </th>
                                    <th class="px-6 py-3 border-b border-
gray-200 bg-gray-50 text-left text-xs leading-4 font-medium text-gray-
500 uppercase tracking-wider">
                                        Email
                                    </th>
                                    <th class="px-6 py-3 border-b border-
gray-200 bg-gray-50"></th>
                                </tr>
                            </thead>
                            <tbody>

```

```

        <tr class="bg-white" th:each="user : ${users}"> ①
            <td class="px-6 py-4 whitespace-no-wrap
text-sm leading-5 font-medium text-gray-900"
th:text="${user.userName.fullName}"> ②
                Bernard Lane
            </td>
            <td class="px-6 py-4 whitespace-no-wrap
text-sm leading-5 text-gray-500" th:text="${user.gender}"> ③
                MALE
            </td>
            <td class="px-6 py-4 whitespace-no-wrap
text-sm leading-5 text-gray-500" th:text="${user.birthday}"> ④
                2000-01-15
            </td>
            <td class="px-6 py-4 whitespace-no-wrap
text-sm leading-5 text-gray-500" th:text="${user.email.asString()}"> ⑤
                bernard.lane@gmail.com
            </td>
            <td class="px-6 py-4 whitespace-no-wrap
text-right text-sm leading-5 font-medium">
                <a href="#" class="text-indigo-600
hover:text-indigo-900">Edit</a>
            </td>
        </tr>
    </tbody>
</table>
</div>
</div>
</div>
</div>
</div>
</div>
</body>
</html>

```

① Use `th:each` to loop over the `users`.

② Output the full name of the `user`.

We need to update `UserName` with the `getFullName()` method to make this work:

`com.tamingthymeleaf.application.user.UserName`

```

public String getFullName() {
    return String.format("%s %s", firstName, lastName);
}

```

}

- ③ Have the gender in the 2nd column of the table.
- ④ Print the birthday date.
- ⑤ Have the email on the last column.

This results in the following page:

NAME	GENDER	BIRTHDAY	EMAIL	
Alberta Jerde	MALE	2003-06-19	alberta.jerde@hotmail.com	Edit
Ned Murphy	FEMALE	1993-09-09	ned.murphy@gmail.com	Edit
Ethelene Haag	FEMALE	1981-11-26	ethelene.haag@gmail.com	Edit
Brendon Schamberger	MALE	1998-10-17	brendon.schamberger@hotmail.com	Edit
Leo Legros	FEMALE	2009-08-23	leo.legros@yahoo.com	Edit
Brenton Ledner	MALE	1989-04-20	brenton.ledner@yahoo.com	Edit
Casimira DuBuque	FEMALE	1986-09-11	casimira.dubuque@yahoo.com	Edit
Leigh Monahan	MALE	1985-12-18	leigh.monahan@gmail.com	Edit
Guy O'Connell	MALE	1981-12-31	guy.oconnell@gmail.com	Edit
Reuben Schinner	MALE	1990-02-13	reuben.schinner@yahoo.com	Edit

Figure 30. Table with users with default rendering

## 10.3. Refactor the table using fragments

This already looks pretty good in the browser, but we have quite some duplication in the `<th>` and `<td>` tags. Let's create a few small fragments to avoid that.

This is how the `<th>` tag is currently used:

```
<tr>
  <th class="px-6 py-3 border-b border-gray-200 bg-gray-50 text-left text-xs leading-4 font-medium text-gray-500 uppercase tracking-wider">
    Name
  </th>
```

```

<th class="px-6 py-3 border-b border-gray-200 bg-gray-50 text-left
text-xs leading-4 font-medium text-gray-500 uppercase tracking-wider">
    Gender
</th>
<th class="px-6 py-3 border-b border-gray-200 bg-gray-50 text-left
text-xs leading-4 font-medium text-gray-500 uppercase tracking-wider">
    Birthday
</th>
<th class="px-6 py-3 border-b border-gray-200 bg-gray-50 text-left
text-xs leading-4 font-medium text-gray-500 uppercase tracking-wider">
    Email
</th>
<th class="px-6 py-3 border-b border-gray-200 bg-gray-50"></th>
</tr>

```

Create a new fragment `table.html` to put in all the table related fragments. Add a fragment `header` that represents how we want to style our table headers:

```

<th th:fragment="header(title)"
     class="px-6 py-3 border-b border-gray-200 bg-gray-50 text-left
text-xs leading-4 font-medium text-gray-500 uppercase tracking-wider"
     th:text="${title}">
    Header title
</th>

```



Be careful *not* to use the name of a HTML tag as the fragment name. Because Thymeleaf can use CSS selectors to match a fragment, it would lead to confusing results. See [Appendix C: Markup Selector Syntax](#) in the Thymeleaf documentation for more details.

The fragments accepts a single argument for the title text of the header. Using this fragment in `users/list.html`, makes the HTML page a lot more readable (Remember to add each of the translation keys to `src/main/resources/i18n/messages.properties`):

```

<tr>
    <th th:replace="fragments/table :: header(#{user.name})"></th>
    <th th:replace="fragments/table :: header(#{user.gender})"></th>
    <th th:replace="fragments/table :: header(#{user.birthday})"></th>
    <th th:replace="fragments/table :: header(#{user.email})"></th>
    <th th:replace="fragments/table :: header('')"></th>
</tr>

```

We can now do the same for the body of the table. This is the current body:

```

<tr class="bg-white" th:each="user : ${users}"> ①
    <td class="px-6 py-4 whitespace-no-wrap text-sm leading-5 font-
medium text-gray-900" th:text="${user.userName.fullName}"> ②
        Bernard Lane
    </td>
    <td class="px-6 py-4 whitespace-no-wrap text-sm leading-5 text-gray-
500" th:text="${user.gender}"> ③
        MALE
    </td>
    <td class="px-6 py-4 whitespace-no-wrap text-sm leading-5 text-gray-
500" th:text="${user.birthday}"> ④
        2000-01-15
    </td>
    <td class="px-6 py-4 whitespace-no-wrap text-sm leading-5 text-gray-
500" th:text="${user.email.asString()}"> ⑤
        bernard.lane@gmail.com
    </td>
    <td class="px-6 py-4 whitespace-no-wrap text-right text-sm leading-5
font-medium">
        <a href="#" class="text-indigo-600 hover:text-indigo-900">
Edit</a>
    </td>
</tr>

```

We have to be careful here, as there are 3 cases:

1. The first `<td>` defines `font-medium text-gray-900` so the text is bigger and bolder there.
2. The "normal" `<td>` for the remaining columns that display data.
3. The last `<td>` column that has a child tag with an `<a>` which will lead to the edit page later.

We can create a single fragment with an optional parameter for the first 2 cases:

```

<td th:fragment="data(contents)"
    th:with="primary=${primary?: false}"
    class="px-6 py-4 whitespace-no-wrap text-sm leading-5 text-
gray-500"
    th:classappend="${primary?'font-medium text-gray-900':''}"
    th:text="${contents}">
    Table data contents
</td>

```

To have a fragment with an optional parameter in Thymeleaf, we can use the `th:with` attribute. In this example, we define `primary` inside `th:with`. If the caller passes in `primary`, we use that value. If not, we default to `false` via the "elvis operator" (`?:`). We then use the value of `primary` to define if

the extra CSS classes need to be added or not.

This is the fragment for the `<td>` that has the link:

```
<td th:fragment="dataWithLink(linkText, linkUrl)"
    class="px-6 py-4 whitespace-no-wrap text-right text-sm
leading-5 font-medium">
    <a th:href="${linkUrl}"
        th:text="${linkText}"
        class="text-indigo-600 hover:text-indigo-900">Edit</a>
</td>
<td>bla</td>
```

Using those fragments in `users/list.html` results in this:

```
<tr class="bg-white" th:each="user : ${users}">
    <td th:replace="fragments/table :: data(contents=${user.userName.fullName},primary=true)"></td> ①
    <td th:replace="fragments/table :: data(${user.gender})"></td>
    <td th:replace="fragments/table :: data(${user.birthday})"></td>
    <td th:replace="fragments/table :: data(${user.email.asString()})"></td>
    <td th:replace="fragments/table :: dataWithLink(#{edit}, '#')"></td>
②
</tr>
```

① We set `primary` to true to have the first column use the bolder styling. Note how we need to provide the `contents` parameter name as well.

② Use the `dataWithLink` fragment, passing in the link text and link URL (which is a placeholder for now)

The resulting rendering in the browser has not changed by this, but our code is now in a much better shape.

## 10.4. Use pagination

We currently load all users from the database in 1 big table and present that. To avoid overloading the database, we should implement pagination so we can get users in batches of 10 or 20 users for example.

Spring Data JPA makes it very easy to implement pagination. We need to change our `UserRepository` to extend `PagingAndSortingRepository` as opposed to `CrudRepository`:

```
package com.tamingthymeleaf.application.user;

import org.springframework.data.repository.PagingAndSortingRepository;
```

```
import org.springframework.transaction.annotation.Transactional;

@Transactional(readOnly = true)
public interface UserRepository extends PagingAndSortingRepository<User,
UserId>, UserRepositoryCustom {
}
```

By doing so, our repository now allows to get entities in pages. From the [PagingAndSortingRepository](#) source code:

```
/** 
 * Returns a {@link Page} of entities meeting the paging restriction
provided in the {@code Pageable} object.
*
* @param pageable
* @return a page of entities
*/
Page<T> findAll(Pageable pageable);
```

[Pageable](#) represents the input parameters that will allow the database to return the correct set of results. [Page](#) represents those results together with some metadata like total number of pages, total number of elements, ...

We can test our new paging capability by writing an extra test in [UserRepositoryTest](#):

```
@Test
void testfindAllPageable() {
    saveUsers(8); ①

    Sort sort = Sort.by(Sort.Direction.ASC, "userName.lastName",
"userName.firstName"); ②
    assertThat(repository.findAll(PageRequest.of(0, 5, sort))) ③
        .hasSize(5) ④
        .extracting(user -> user.getUserName().getFullName()) ⑤
        .containsExactly("Tommy1 Holt", "Tommy3 Holt", "Tommy5
Holt", "Tommy7 Holt", "Tommy0 Walton"); ⑥

    assertThat(repository.findAll(PageRequest.of(1, 5, sort))) ⑦
        .hasSize(3)
        .extracting(user -> user.getUserName().getFullName())
        .containsExactly("Tommy2 Walton", "Tommy4 Walton",
"Tommy6 Walton");

    assertThat(repository.findAll(PageRequest.of(2, 5, sort
```

```

)).isEmpty(); ⑧
}

private void saveUsers(int numberOfWorkers) {
    for (int i = 0; i < numberOfWorkers; i++) {
        repository.save(new User(repository.nextId(),
            new UserName(String.format
("Tommy%d", i), i % 2 == 0 ? "Walton" : "Holt"),
            Gender.MALE,
            LocalDate.of(2001, Month.FEBRUARY,
17),
            new Email("tommy.walton" + i +
"@gmail.com"),
            new PhoneNumber("202 555 0192")));
    }
}

```

- ① Save 8 users in the database to set up the test.
- ② In order to have paging properly work, you need to always assign a sort order. Otherwise, you can never be sure of what part of the data the database will return. Here, we set up an ascending sort on last name and first name.
- ③ Pages start at 0, so we are requesting the first page with a page size of 5.
- ④ Assert if the returned page has 5 elements (Since we saved 8, we know the first page needs to have 5).
- ⑤ Extract the full name from the user, so we can assert if the sorting was correct.
- ⑥ Assert if the names of the users are in the expected order.
- ⑦ Ask for the 2nd page. This should have the 3 remaining users.
- ⑧ Ask for the 3rd page. This should be empty.

Run the `UserRepositoryTest`, all should be green. The SQL logging will also show that the sorting is applied on the database level:

```

select user0_.id as id1_0_, user0_.birthday as birthday2_0_,
user0_.email as email3_0_, user0_.gender as gender4_0_,
user0_.phone_number as phone_nu5_0_, user0_.first_name as first_na6_0_,
user0_.last_name as last_nam7_0_ from tt_user user0_ order by
user0_.last_name asc, user0_.first_name asc limit ?

```

Next up, we change `UserService` to use our new `findAll(Pageable)` method on the `UserRepository`. We replace `ImmutableSet<User> getAllUsers()` with `getUsers(Pageable)`:

```

package com.tamingthymeleaf.application.user;

import org.springframework.data.domain.Page;

```

```

import org.springframework.data.domain.Pageable;

public interface UserService {
    User createUser(CreateUserParameters parameters);

    Page<User> getUsers(Pageable pageable); ①
}

```

① The return type is now a `org.springframework.data.domain.Page` instead of an `ImmutableSet` we had before.

The implementation in `UserServiceImpl` just delegates to the repository:

```

@Override
public Page<User> getUsers(Pageable pageable) {
    return repository.findAll(pageable);
}

```

We can now update the controller to make use of the new service method:

```

package com.tamingthymeleaf.application.user.web;

import com.tamingthymeleaf.application.user.UserService;
import org.springframework.data.domain.Pageable;
import org.springframework.data.web.SortDefault;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
@RequestMapping("/users")
public class UserController {

    private final UserService service;

    public UserController(UserService service) {
        this.service = service;
    }

    @GetMapping
    public String index(Model model,
                        @SortDefault.SortDefaults({
                            @SortDefault("userName.lastName"),

```

```

    @SortDefault("userName.firstName"))
Pageable pageable) { ①
    model.addAttribute("users", service.getUsers(pageable)); ②
    return "users/list";
}
}

```

- ① We add an extra parameter of type `Pageable` with the `@SortDefault` annotation to set the default sort order. Spring MVC will inject a correct instance of the `Pageable` object depending on the query parameters used.

Before we test this, we can set the global page size Spring Data should use:

*application.properties*

```
spring.data.web.pageable.default-page-size=10
```

Running the application, we now get a table with the first 10 users, sorted by last name (and first name):

NAME	GENDER	BIRTHDAY	EMAIL	
Rosario Barton	MALE	2005-06-24	rosario.barton@yahoo.com	Edit
Raeann Boehm	MALE	1994-02-18	raeann.boehm@gmail.com	Edit
Matha Bradtke	MALE	1980-11-20	matha.bradtke@hotmail.com	Edit
Casimira DuBuque	FEMALE	1986-09-11	casimira.dubuque@yahoo.com	Edit
Rigoberto Feeney	MALE	1995-06-09	rigoberto.feeney@hotmail.com	Edit
Sau Ferry	MALE	1995-04-15	sau.ferry@yahoo.com	Edit
Shannan Gaylord	MALE	1996-03-18	shannan.gaylord@hotmail.com	Edit
Clair Gislason	MALE	1994-10-04	clair.gislason@hotmail.com	Edit
Ethelene Haag	FEMALE	1981-11-26	ethelene.haag@gmail.com	Edit
Alberta Jerde	MALE	2003-06-19	alberta.jerde@hotmail.com	Edit

Figure 31. Table with users (first page)

We can get to the other pages by manipulating the URL to add the `page` query parameter:

NAME	GENDER	BIRTHDAY	EMAIL	
Brenton Ledner	MALE	1989-04-20	brenton.ledner@yahoo.com	<a href="#">Edit</a>
Leo Legros	FEMALE	2009-08-23	leo.legros@yahoo.com	<a href="#">Edit</a>
Leigh Monahan	MALE	1985-12-18	leigh.monahan@gmail.com	<a href="#">Edit</a>
Ned Murphy	FEMALE	1993-09-09	ned.murphy@gmail.com	<a href="#">Edit</a>
Guy O'Connell	MALE	1981-12-31	guy.oconnell@gmail.com	<a href="#">Edit</a>
Michal Rogahn	MALE	1988-08-11	michal.rogahn@gmail.com	<a href="#">Edit</a>
Brendon Schamberger	MALE	1998-10-17	brendon.schamberger@hotmail.com	<a href="#">Edit</a>
Reuben Schinner	MALE	1990-02-13	reuben.schinner@yahoo.com	<a href="#">Edit</a>
Erin Schuster	FEMALE	1989-06-17	erin.schuster@hotmail.com	<a href="#">Edit</a>
Jerald Welch	FEMALE	1994-07-26	jerald.welch@gmail.com	<a href="#">Edit</a>

Figure 32. Second page by adding `?page=1` to the URL

Manually manipulating the URL is obviously bad UX, so let's add some pagination controls to our UI.

We can start by copying the HTML of one of the [pagination controls of Tailwind UI](#) into `fragments/pagination.html`. Using the fragment for our list of users gives us:

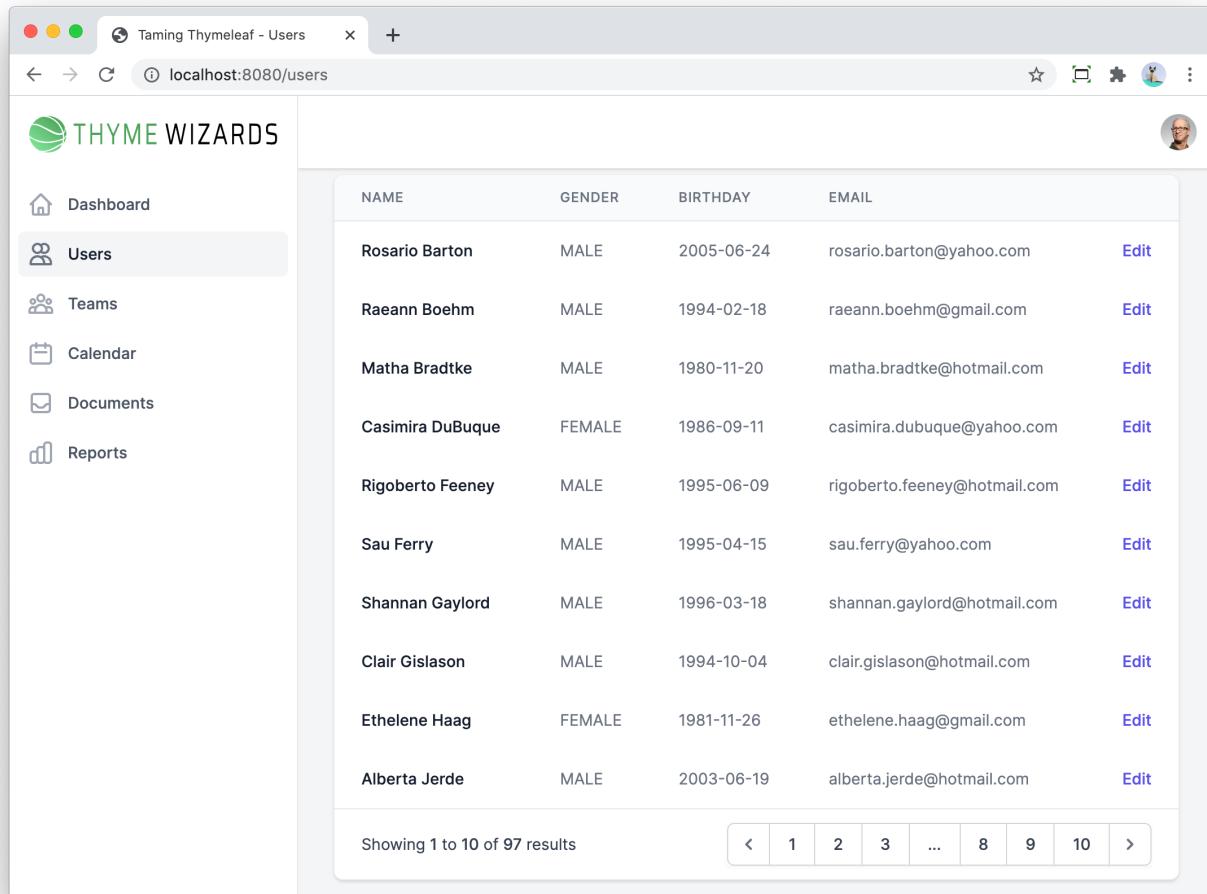


Figure 33. UI mockup of pagination

This already looks great, but it doesn't work yet. We need to implement the following behaviour in our pagination fragment:

- Show the correct number of pages
- Disable the previous button if on the first page
- Disable the next button if on the last page
- Highlight the current page number
- Show a few pages before and after the current page, hiding the other page numbers.

In order to build a fully re-usable pagination component, the fragment will take a `page` variable of the type `org.springframework.data.domain.Page`. We will also use the `org.springframework.web.servlet.support.ServletUriComponentsBuilder` class from Spring to build the URLs for each page button. Let's go through the code bit by bit.

At the top-level of the fragment, we define `urlBuilder` variable representing the `ServletUriComponentsBuilder` class so we can call static methods on that class:

```
<div th:fragment="controls"
      class="bg-white px-4 py-3 flex items-center justify-between border-t border-gray-200 sm:px-6">
```

```
th:with="urlBuilder=${T(org.springframework.web.servlet.support.ServletU
riComponentsBuilder)}">
```

That top-level `<div>` has 2 child tags: one for mobile and one for desktop.

For mobile, we only have a 'Previous' and 'Next' button with no indication of the current page. We use the `page.isFirst()` method to conditionally enable or disable the 'Previous' button. For the 'Next' button, we use `page.isLast()`.

```
<div class="flex-1 flex justify-between sm:hidden">
    <a id="pagination-mobile-previous"

        th:href="${page.isFirst()?'javascript:void(0)':${urlBuilder.fromCurrent
        Request().replaceQueryParam('page', page.number - 1).toUriString()}}"
            class="relative inline-flex items-center px-4 py-2 border
        text-sm leading-5 font-medium rounded-md"
                th:classappend="${page.isFirst()?'pointer-events-none text-
        gray-200 border-gray-200':'border-gray-300 text-gray-700 bg-white
        hover:text-gray-500 focus:outline-none focus:shadow-outline-blue
        focus:border-blue-300 active:bg-gray-100 active:text-gray-700 transition
        ease-in-out duration-150'}"
                th:disabled="${page.isFirst()}"
                th:text="#{pagination.previous}">
                    Previous
                </a>
                <a id="pagination-mobile-next"

        th:href="${page.isLast()?'javascript:void(0)':${urlBuilder.fromCurrentR
        equest().replaceQueryParam('page', page.number + 1).toUriString()}}"
            class="ml-3 relative inline-flex items-center px-4 py-2
        border text-sm leading-5 font-medium rounded-md"
                th:classappend="${page.isLast()?'pointer-events-none text-
        gray-200 border-gray-200':'border-gray-300 text-gray-700 bg-white
        hover:text-gray-500 focus:outline-none focus:shadow-outline-blue
        focus:border-blue-300 active:bg-gray-100 active:text-gray-700 transition
        ease-in-out duration-150'}"
                th:disabled="${page.isLast()}"
                th:text="#{pagination.next}">
                    Next
                </a>
    </div>
```

Make particular note to how the `href` is build via the `urlBuilder` variable:

- `urlBuilder.fromCurrentRequest()` gives us the current browser request

- `.replaceQueryParam('page', page.number - 1)` replaces the current query parameter `page` with the current page number (`page.number`) minus 1 (for the 'Previous' button).
- `.toUriString()` returns the new URL to use for the `href`.



In case you are wondering: `replaceQueryParam` also works if the param is not there yet. It will be added in that case to the URL.

For desktop, we have the following parts:

- The pagination summary text on the left side (*Showing x to y of z results*)
- The previous/next arrows
- The page buttons

The pagination summary part is fairly straight forward using the `Page` methods `getSize()`, `getNumber()`, `getNumberOfElements()` and `getTotalElements()`:

```
<div>
    <p id="pagination-summary" class="text-sm leading-5 text-gray-700">
        Showing
        <span class="font-medium" th:text="${(page.getSize() * page.getNumber()) + 1}">1</span>
        to
        <span class="font-medium" th:text="${(page.getSize() * page.getNumber()) + page.getNumberOfElements()}">10</span>
        of
        <span class="font-medium"
th:text="${page.getTotalElements()}">97</span>
        results
    </p>
</div>
```

This solution glosses over the details of providing proper translation of the pagination summary. We should not create translation keys for *Showing*, *to*, *of* and *results* parts separately since we cannot be sure the order will be the same in other languages.

One possible solution is to add the `<span>` tags to the actual translation to keep the styling:



```
pagination.summary=Showing <span class="font-medium">{0}</span> to <span class="font-medium">{1}</span>
of <span class="font-medium">{2}</span> results
```

`{0}`, `{1}` and `{2}` are placeholders where the actual values can be passed in from the HTML:

```
<p id="pagination-summary" class="text-sm leading-5 text-gray-700"
    th:with="firstRowNum=${page.getSize() * page.getNumber() + 1},lastRowNum=${(page.getSize() * page.getNumber() + page.getTotalElements())}"
    th:utext="#{pagination.summary(${firstRowNum}, ${lastRowNum}, ${page.getTotalElements()})}">
</p>
```

We need to use `th:utext` (as opposed to `th:text`) to avoid that Thymeleaf would escape the `<span>` tag.

The previous/next arrows are very similar to the mobile versions. this is the source for the 'Previous' button:

```
<a id="pagination-previous"

th:href="${page.isFirst()?'javascript:void(0)':${urlBuilder.fromCurrentRequest().replaceQueryParam('page', page.number - 1).toUriString()}}"
        class="relative inline-flex items-center px-2 py-2 rounded-l-md border bg-white text-sm leading-5 font-medium"
        th:aria-label="#{pagination.previous}"
        th:classappend="${page.isFirst()?'pointer-events-none text-gray-200 border-gray-200':'border-gray-300 text-gray-500 hover:text-gray-400 focus:z-10 focus:outline-none focus:border-blue-300 focus:shadow-outline-blue active:bg-gray-100 active:text-gray-500 transition ease-in-out duration-150'}"
        th:disabled="${page.isFirst()}">
    <svg class="h-5 w-5" fill="currentColor" viewBox="0 0 20 20">
        <path fill-rule="evenodd" d="M12.707 5.293a1 1 0 010 1.414L9.414 10l3.293 3.293a1 1 0 01-1.414 1.414l-4-4a1 1 0 010-1.414l4-4a1 1 0 011.414 0z" clip-rule="evenodd"/>
    </svg>
</a>
```

The most tricky part is the page buttons implementation:

```
<th:block
    th:with="startPage=${T(Math).max(1, page.getNumber() - 1)},endPage=${T(Math).min(startPage + 4, page.getTotalPages())}">
```

```

<a th:each="pageNumber :
${#numbers.sequence(startPage, endPage)}"
    th:id="${'pagination-page-' + pageNumber}">

th:href="${urlBuilder.fromCurrentRequest().replaceQueryParam('page',
pageNumber - 1).toUriString()}"
    class="-ml-px relative inline-flex items-center
px-4 py-2 border border-gray-300 bg-white text-sm leading-5 font-medium
text-gray-700 hover:text-gray-500 focus:z-10 focus:outline-none
focus:border-blue-300 focus:shadow-outline-blue active:bg-gray-100
active:text-gray-700 transition ease-in-out duration-150"
    th:classappend="${page.number == pageNumber -
1?'font-bold':''}">
    <span th:text="${pageNumber}" th:remove=
"tag"></span>
    </a>
</th:block>

```

We use a `th:block` here to generate a maximum of 5 buttons around the current page number, taking into account that we cannot go below 1 or above the total number of pages. The result of those calculations are put in the `startPage` and `endPage` variables in the `th:with` attribute. Using those variables, we iterate over a sequence of numbers from `startPage` to `endPage` to generate the buttons. Thymeleaf has the built-in `#numbers.sequence(start,end)` to do that. Also note how we bold the current page number in the `th:classappend` attribute.

With our pagination fragment complete, all that is left is to call it with the appropriate variable in `users/list.html`:

```
<div th:replace="fragments/pagination :: controls(page=${users})"></div>
```

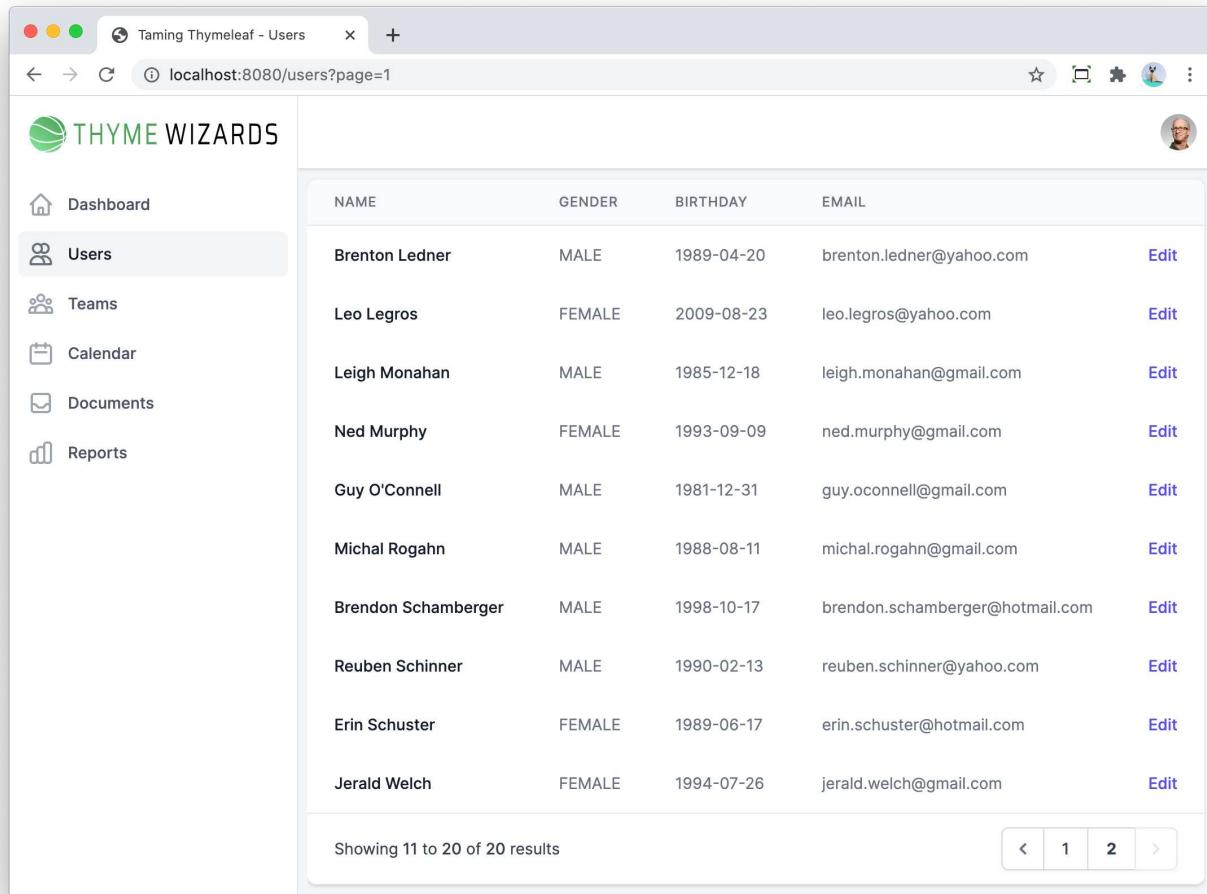
If we run this, we can see that the pagination is working properly:

The screenshot shows a web application interface titled "Taming Thymeleaf - Users" at the URL "localhost:8080/users". The left sidebar contains navigation links: Dashboard, Users (which is selected), Teams, Calendar, Documents, and Reports. The main content area has a header "THYME WIZARDS" and a user profile picture. A table displays user data with columns: NAME, GENDER, BIRTHDAY, and EMAIL. Each row includes an "Edit" link. The table shows 10 results out of 20. At the bottom, there is a pagination control with buttons for <, 1, 2, and >.

NAME	GENDER	BIRTHDAY	EMAIL	
Rosario Barton	MALE	2005-06-24	rosario.barton@yahoo.com	Edit
Raeann Boehm	MALE	1994-02-18	raeann.boehm@gmail.com	Edit
Matha Bradtke	MALE	1980-11-20	matha.bradtke@hotmail.com	Edit
Casimira DuBuque	FEMALE	1986-09-11	casimira.dubuque@yahoo.com	Edit
Rigoberto Feeney	MALE	1995-06-09	rigoberto.feeney@hotmail.com	Edit
Sau Ferry	MALE	1995-04-15	sau.ferry@yahoo.com	Edit
Shannan Gaylord	MALE	1996-03-18	shannan.gaylord@hotmail.com	Edit
Clair Gislason	MALE	1994-10-04	clair.gislason@hotmail.com	Edit
Ethelene Haag	FEMALE	1981-11-26	ethelene.haag@gmail.com	Edit
Alberta Jerde	MALE	2003-06-19	alberta.jerde@hotmail.com	Edit

Showing 1 to 10 of 20 results

Figure 34. Pagination fragment working



The screenshot shows a web browser window titled "Taming Thymeleaf - Users". The URL in the address bar is "localhost:8080/users?page=1". The page itself is titled "THYME WIZARDS" and features a sidebar with links: Dashboard, Users (which is active), Teams, Calendar, Documents, and Reports. The main content area displays a table of user data with columns: NAME, GENDER, BIRTHDAY, and EMAIL. The table contains 10 rows of data. At the bottom of the table, it says "Showing 11 to 20 of 20 results". To the right of the table is a navigation bar with icons for back, forward, search, and other functions. A small profile picture is visible in the top right corner of the main content area.

NAME	GENDER	BIRTHDAY	EMAIL	
Brenton Ledner	MALE	1989-04-20	brenton.ledner@yahoo.com	<a href="#">Edit</a>
Leo Legros	FEMALE	2009-08-23	leo.legros@yahoo.com	<a href="#">Edit</a>
Leigh Monahan	MALE	1985-12-18	leigh.monahan@gmail.com	<a href="#">Edit</a>
Ned Murphy	FEMALE	1993-09-09	ned.murphy@gmail.com	<a href="#">Edit</a>
Guy O'Connell	MALE	1981-12-31	guy.oconnell@gmail.com	<a href="#">Edit</a>
Michal Rogahn	MALE	1988-08-11	michal.rogahn@gmail.com	<a href="#">Edit</a>
Brendon Schamberger	MALE	1998-10-17	brendon.schamberger@hotmail.com	<a href="#">Edit</a>
Reuben Schinner	MALE	1990-02-13	reuben.schinner@yahoo.com	<a href="#">Edit</a>
Erin Schuster	FEMALE	1989-06-17	erin.schuster@hotmail.com	<a href="#">Edit</a>
Jerald Welch	FEMALE	1994-07-26	jerald.welch@gmail.com	<a href="#">Edit</a>

Figure 35. Pagination if page 2 is active

If you want to test the pagination some more, you can append `size=3` to the URL to set the page size. This will give you more pages for testing:



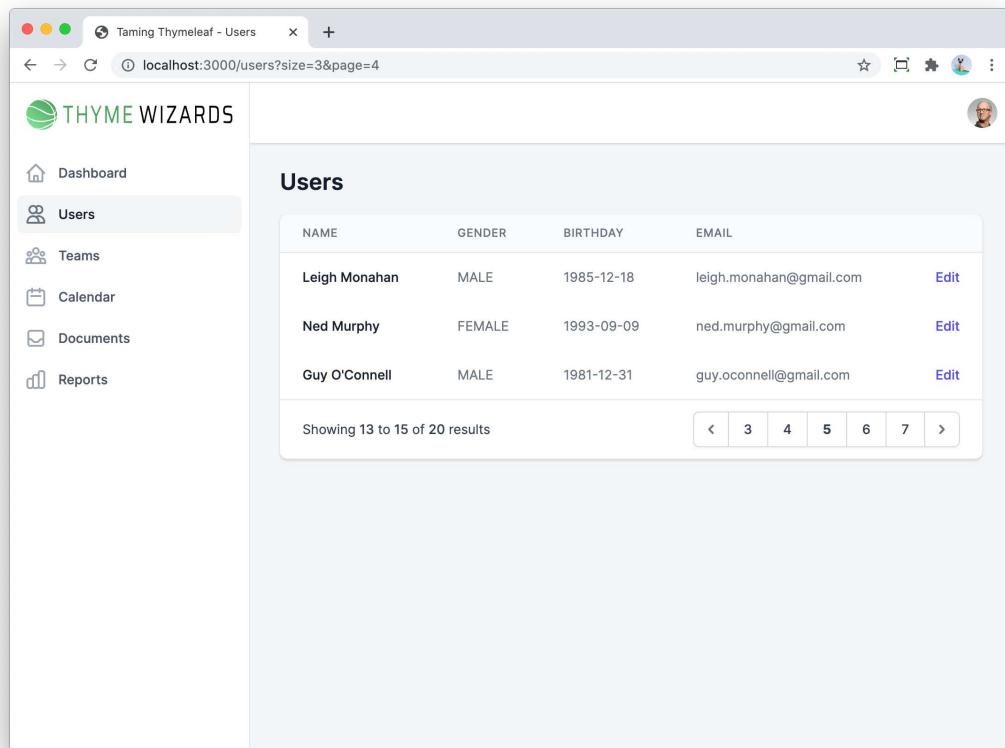


Figure 36. Pagination with 3 users per page

## 10.5. Hide columns on mobile

The menu on the left hides into a hamburger menu on mobile, but our table keeps showing the 4 columns (+1 for the edit links).

If we only want to keep the 'Name' and 'Edit' columns, we should hide the other columns using Tailwind. If we follow the recommendation on [Targeting mobile screens](#), we should add the `hidden sm:table-cell` styles to the columns we want to hide. This will ensure they are hidden on the small screen and become visible as soon as the screen is bigger.

This is the adjusted `header` fragment with a new parameter `hideOnMobile` to add the extra CSS classes:

```
<th th:fragment="header(title)"
    th:with="hideOnMobile=${hideOnMobile?:false}"
    class="px-6 py-3 border-b border-gray-200 bg-gray-50 text-left text-xs leading-4 font-medium text-gray-500 uppercase tracking-wider"
    th:classappend="${hideOnMobile?'hidden sm:table-cell':''}"
    th:text="#">{title}
```

Header title

And this is how we can use it:

```
<tr>
    <th th:replace="fragments/table :: header(#{user.name})"></th>
    <th th:replace="fragments/table :: header(title=#{user.gender},hideOnMobile=true)"></th>
    <th th:replace="fragments/table :: header(title=#{user.birthday},hideOnMobile=true)"></th>
    <th th:replace="fragments/table :: header(title=#{user.email},hideOnMobile=true)"></th>
    <th th:replace="fragments/table :: header('')"></th>
</tr>
```

Similar for the `data` fragment:

```
<td th:fragment="data(contents)"
    th:with="primary=${primary?:
false},hideOnMobile=${hideOnMobile?:false}"
    class="px-6 py-4 whitespace-no-wrap text-sm leading-5 text-gray-500"
    th:classappend="|${primary?'font-medium text-gray-900':''}
${hideOnMobile?'hidden sm:table-cell':''}|"
    th:text="${contents}">
    Table data contents
</td>
```

Note the [Literal substitutions](#) syntax to be able to have multiple conditions in the `th:classappend` attribute.

Using the updated fragment:

```
<tr class="bg-white" th:each="user : ${users}">
    <td th:replace="fragments/table :: data(contents=${user.userName.fullName},primary=true)"></td> ①
    <td th:replace="fragments/table :: data(contents=${user.gender},hideOnMobile=true)"></td>
    <td th:replace="fragments/table :: data(contents=${user.birthday},hideOnMobile=true)"></td>
    <td th:replace="fragments/table :: data(contents=${user.email.asString()},hideOnMobile=true)"></td>
    <td th:replace="fragments/table :: dataWithLink('Edit', '#')"></td>
②
</tr>
```

This is rendered on a mobile device as follows:



Rosario Barton	Edit
Raeann Boehm	Edit
Matha Bradtke	Edit
Casimira DuBuque	Edit
Rigoberto Feeney	Edit
Sau Ferry	Edit
Shannan Gaylord	Edit
Clair Gislason	Edit
Ethelene Haag	Edit
Alberta Jerde	Edit

[Previous](#) [Next](#)

Figure 37. Mobile UI with hidden columns

## 10.6. Summary

In this chapter, you learned:

- How to show a list of entities on a HTML page.
- How to use pagination to page through a long list of results.
- How to make a HTML table adapt to different screen sizes.

# Chapter 11. Forms

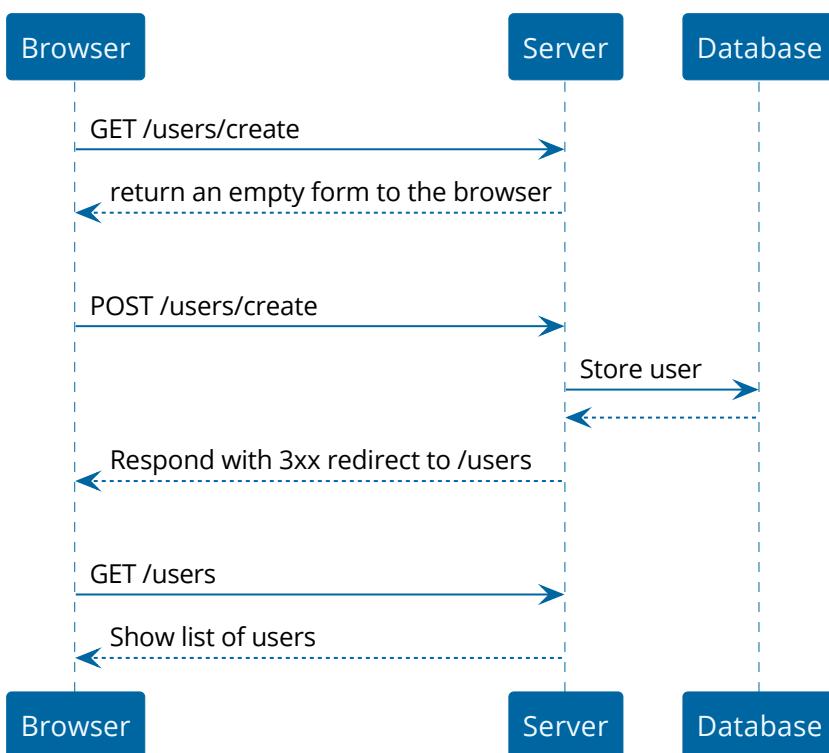
## 11.1. Form fields

As an example on how to use HTML forms, we will implement creating a user in the UI.

Implementing a form submit is a multi-phase process:

1. The browser does a **GET** to display the form.
2. The user enters information using the form elements.
3. The browser does a **POST** with the form elements' information.
4. If there are no validation errors, the browser gets redirected to avoid double submissions.
5. If there are validation errors, the form remains in place so the user can correct the information.

This diagram shows the success flow of the process:



As a first implementation step, we need to create an object that will map each HTML form input to a property of the Java object:

```

package com.tamingthymeleaf.application.user.web;

import com.tamingthymeleaf.application.user.CreateUserParameters;
import com.tamingthymeleaf.application.user.Gender;
import com.tamingthymeleaf.application.user.PhoneNumber;
import com.tamingthymeleaf.application.user.UserName;
import org.springframework.format.annotation.DateTimeFormat;
  
```

```

import javax.validation.constraints.Email;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Pattern;
import java.time.LocalDate;

public class CreateUserData {
    @NotBlank
    private String firstName;
    @NotBlank
    private String lastName;
    @NotNull
    private Gender gender;
    @NotBlank
    @Email
    private String email;
    @NotNull
    @DateTimeFormat(pattern = "yyyy-MM-dd")
    private LocalDate birthday;
    @NotBlank
    @Pattern(regexp = "[0-9.\\-() x/+]+")
    private String phoneNumber;

    // Getters and setters omitted

}

```

Compared to the `CreateUserParameters` object that uses rich value objects, we restrict ourselves here to mainly `String` types and no nesting (like with `UserName`). This will make it easier to map the fields of the form to the `CreateUserData` object.

We see 2 exceptions that are not `String`:

- `gender`: This is an `enum` and will work out-of-the-box.
- `birthday`: This is a `LocalDate` and needs the `DateTimeFormat` annotation to indicate how the date will be present in the HTML form input.

The annotations `@NotBlank`, `@NotNull`, `@Email` and `@Pattern` will validate the input we receive from the form. This is what is called *server-side validation*, which is the type of validation that you always need to perform since you cannot trust if *client-side validation* has actually happened.

If you are wondering if you need both types of validation, I usually think of it this way: client-side validation is needed to have better usability, server-side validation is needed to protect the application from invalid data.

We can now implement the `GET` method in `UserController`:

`com.tamingthymeleaf.application.user.web.UserController`

```
@GetMapping("/create") ①
public String createUserForm(Model model) { ②
    model.addAttribute("user", new CreateUserFormData()); ③
    model.addAttribute("genders", List.of(Gender.MALE, Gender.
FEMALE, Gender.OTHER)); ④
    return "users/edit"; ⑤
}
```

- ① Bind the method to a `GET` request on the `/users/create` url.
- ② Inject the `Model` instance.
- ③ Add an empty `CreateUserFormData` object to the model under the `user` key.
- ④ Add the list of possible genders. This will be used to generate a radio button for each option.
- ⑤ Return the path to the Thymeleaf template that will render the form.

With our controller updated, we can now add our Thymeleaf template that contains the form input controls. This is the full source of `edit.html` so you can get an overview of the code. We will break it down piece by piece below the code block.

`src/main/resources/templates/users/edit.html`

```
<!DOCTYPE html>
<html
    xmlns:th="http://www.thymeleaf.org"
    xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
    layout:decorate="~{layout/layout}"
    th:with="activeMenuItem='users'>

<head>
    <title>Users</title>
</head>
<body>
<div layout:fragment="page-content">
    <div class="max-w-7xl mx-auto px-4 sm:px-6 md:px-8">
        <h1 class="text-2xl font-semibold text-gray-900"
            th:text="#{user.create}">Create user</h1>
    </div>
    <div class="max-w-7xl mx-auto px-4 sm:px-6 md:px-8">
        <div class="py-4">
            <div class="bg-white shadow px-4 py-5 sm:rounded-lg sm:p-6">
                <form id="user-form"
                    th:object="${user}"
                    th:action="@{/users/create}"
                    method="post">
                    <div>
```

```

<div class="mt-6 grid grid-cols-1 gap-y-6 gap-x-4 sm:grid-cols-6">
    <div class="sm:col-span-6">
        <label class="block text-sm font-medium text-gray-700"
            th:text="#{user.gender}">
            Gender
        </label>
        <div>
            <th:block
                th:each="possibleGender,iter : ${genders}">
                <input type="radio"
                    th:id="${'gender-' + possibleGender}"
                    th:field="*{gender}">
                <br/>
                <div th:value="${possibleGender}">
                    <span class="mr-1 focus:ring-green-500 h-4 w-4 text-green-500 border-gray-300"></span>
                </div>
                <div th:classappend="${iter.index > 0 ? 'ml-4':''}">
                    > <!-- mr-1 transition duration-150 ease-in-out sm:text-sm sm:leading-5 text-green-500 focus:shadow-outline-green-->
                <label th:for="${'gender-' + possibleGender}">
                    th:text="#{'Gender.' + ${possibleGender}}"
                    class="text-sm font-medium text-gray-700">
                    <br/>
                    <!-- sm:text-sm sm:leading-5 -->
                </label>
            </th:block>
        </div>
    </div>

    <div class="sm:col-span-3">
        <label for="firstName" class="block text-sm font-medium text-gray-700"
            th:text="#{user.firstName}">
            First name
        </label>
        <div class="mt-1 rounded-md shadow-sm">

```

```
<input id="firstName"
       type="text"
       th:field="*{firstName}"
       class="shadow-sm focus:ring-
green-500 focus:border-green-500 block w-full sm:text-sm border-gray-300
rounded-md">
</div>
</div>

<div class="sm:col-span-3">
    <label for="lastName" class="block text-
sm font-medium text-gray-700">
        th:text="#{user.lastName}">
        Last name
    </label>
    <div class="mt-1 rounded-md shadow-sm">
        <input id="lastName"
               type="text"
               th:field="*{lastName}"
               class="shadow-sm focus:ring-
green-500 focus:border-green-500 block w-full sm:text-sm border-gray-300
rounded-md">
        </div>
    </div>

    <div class="sm:col-span-4">
        <label for="email" class="block text-sm
font-medium leading-5 text-gray-700">
            th:text="#{user.email}">
            Email address
        </label>
        <div class="mt-1 rounded-md shadow-sm">
            <input id="email"
                   type="email"
                   th:field="*{email}"
                   class="shadow-sm focus:ring-
green-500 focus:border-green-500 block w-full sm:text-sm border-gray-300
rounded-md">
            </div>
        </div>
        <div class="sm:col-span-4">
            <label for="phoneNumber" class="block
text-sm font-medium leading-5 text-gray-700">
                th:text="#{user.phoneNumber}">
                Phone number
            </label>
        </div>
    </div>
</div>
```

```

        </label>
        <div class="mt-1 rounded-md shadow-sm">
            <input id="phoneNumber"
                type="text"
                th:field="*{phoneNumber}"
                class="shadow-sm focus:ring-
green-500 focus:border-green-500 block w-full sm:text-sm border-gray-300
rounded-md">
                </div>
            </div>
            <div class="sm:col-span-2"></div>
            <div class="sm:col-span-2">
                <label for="birthday" class="block text-
sm font-medium leading-5 text-gray-700"
                    th:text="#{user.birthday}">
                    Birthday
                </label>
                <div class="mt-1 rounded-md shadow-sm">
                    <input id="birthday"
                        type="text"
                        th:field="*{birthday}"

th:placeholder="#{user.birthday.placeholder}"
                        class="shadow-sm focus:ring-
green-500 focus:border-green-500 block w-full sm:text-sm border-gray-300
rounded-md">
                    </div>
                </div>
            </div>
            <div class="mt-8 border-t border-gray-200 pt-5">
                <div class="flex justify-end">
<span class="inline-flex rounded-md shadow-sm">
                <button type="button"
                    class="bg-white py-2 px-4 border border-gray-300
rounded-md shadow-sm text-sm font-medium text-gray-700 hover:bg-gray-50
focus:outline-none focus:ring-2 focus:ring-offset-2 focus:ring-green-
500"
                    th:text="#{cancel}">
                    Cancel
                </button>
            </span>
                <span class="ml-3 inline-flex rounded-md
shadow-sm">
                    <button type="submit"

```

```

        class="ml-3 inline-flex justify-center py-2 px-4 border
border-transparent shadow-sm text-sm font-medium rounded-md text-white
bg-green-600 hover:bg-green-700 focus:outline-none focus:ring-2
focus:ring-offset-2 focus:ring-green-500"
        th:text="#{save}">
    Save
</button>
</span>
        </div>
    </div>
</form>
</div>
</div>
</div>
</body>
</html>

```

Let's take it apart piece by piece:

```

<html
    xmlns:th="http://www.thymeleaf.org"
    xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
    layout:decorate="~{layout/layout}"
    th:with="activeMenuItem='users'">

```

At the top of the page, we define the layout we use, just as we did for `users/list.html`. This ensures we have the menu on the left side and the avatar on the top present when rendering this page.

```

<form id="user-form"
    th:object="${user}"
    th:action="@{/users/create}"
    method="post">

```

The `<form>` element itself has 2 important attributes:

- **th:object**: This defines the binding object for the form elements. `user` has to match with the name we used in the controller when we added the `CreateUserFormData` instance to the `model`.
- **th:action**: This indicates the URL that the form will `POST` to.

```

<div class="sm:col-span-3">
    <label for="firstName" class="block text-sm font-medium text-gray-
700">

```

```

        th:text="#{user.firstName}">
    First name
</label>
<div class="mt-1 rounded-md shadow-sm">
    <input id="firstName"
        type="text"
        th:field="*{firstName}"
        class="shadow-sm focus:ring-green-500 focus:border-green-500 block w-full sm:text-sm border-gray-300 rounded-md">
    </div>
</div>

```

Each of the inputs looks similar to the one above for `firstName`. The most important part here is the `th:field="*{firstName}` attribute which binds the value of the HTML input to the `firstName` property of the `CreateUserData` instance. Note the `*{}` syntax used. See [Selected objects](#) for a refresher how this works exactly if needed.

The other input fields are very similar, except for the radio button group to select the gender:

```

<div class="sm:col-span-6">
    <label class="block text-sm font-medium text-gray-700"
        th:text="#{user.gender}">
        Gender
    </label>
    <div>
        <th:block th:each="possibleGender,iter : ${genders}">
            <input type="radio"
                th:id="${'gender-' + possibleGender}"
                th:field="*{gender}"
                th:value="${possibleGender}"
                class="mr-1 focus:ring-green-500 h-4 w-4 text-green-500 border-gray-300"
                th:classappend="${iter.index > 0 ? 'ml-4':''}"
            > <!-- mr-1 transition duration-150 ease-in-out sm:text-sm sm:leading-5 text-green-500 focus:shadow-outline-green-->
            <label th:for="${'gender-' + possibleGender}">
                th:text="${'Gender.' + ${possibleGender}}"
                class="text-sm font-medium text-gray-700">
                <!-- sm:text-sm sm:leading-5 -->
            </label>

        </th:block>
    </div>
</div>

```

We iterate over the `genders` list that was put in the `model` to generate an `<input>` tag for each possible gender. We give each `<input>` a unique id, based on the `name()` of the `Gender` enum. For each `<input>`, we have a corresponding `<label>` so that we can properly style the text next to the radio button.

The `th:classappend` checks the current iteration index to add a left margin to each element, except the first.

If we run the application and go to <http://localhost:8080/users/create>, we should see:

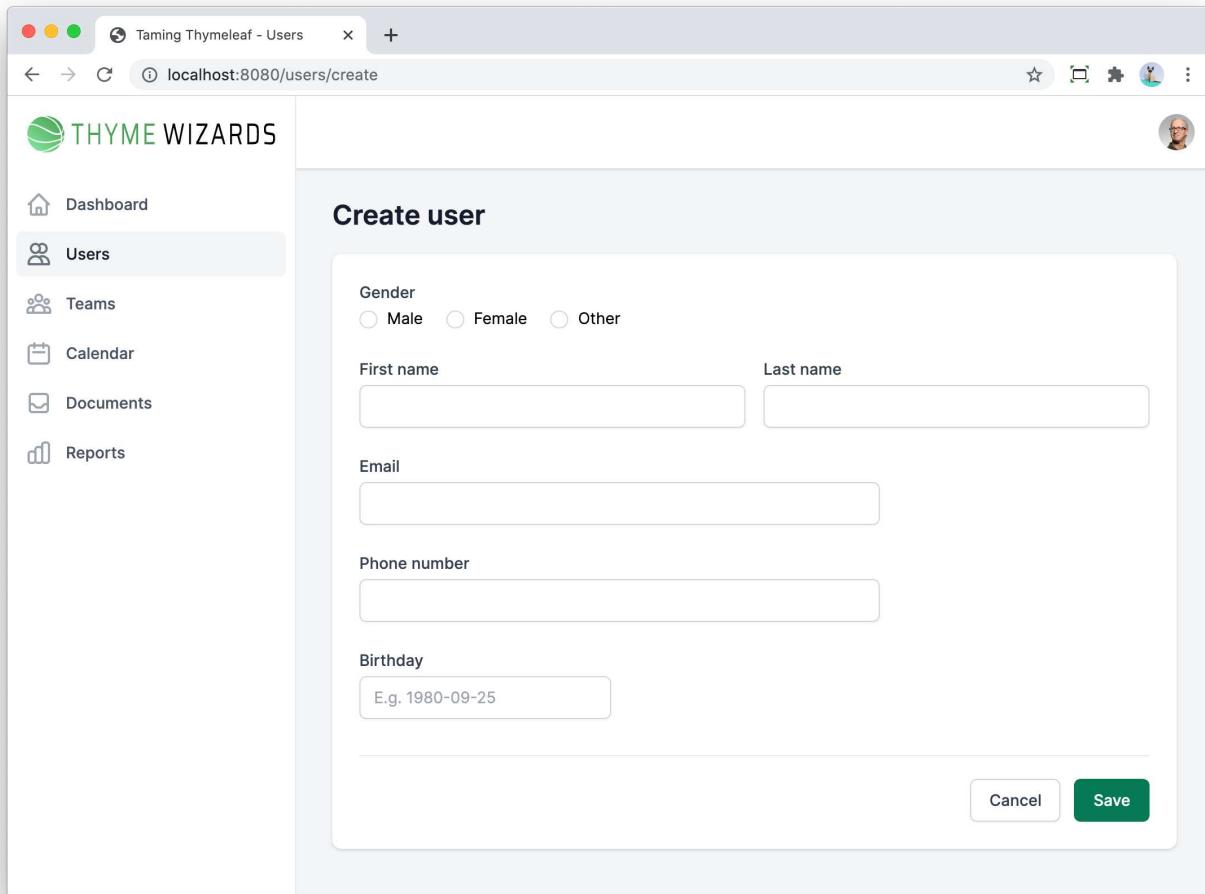


Figure 38. Create user form

To ensure the 'Save' button actually works, we need to implement handling the `POST` in `UserController`:

`com.tamingthymeleaf.application.user.web.UserController`

```

    @PostMapping("/create") ①
    public String doCreateUser(@Valid @ModelAttribute("user")
CreateUserData formData, ②
                           BindingResult bindingResult, Model model)
{ ③
    if (bindingResult.hasErrors()) { ④
        model.addAttribute("genders", List.of(Gender.MALE, Gender
.FEMALE, Gender.OTHER)); ⑤
    }
}

```

```
    return "users/edit"; ⑥
}

service.createUser(formData.toParameters()); ⑦

return "redirect:/users"; ⑧
}
```

- ① Bind the method to a `POST` request on the `/users/create` url.
- ② Inject the `CreateUserFormData` instance that has been put in the model under the `user` key. This one will have the values from the HTML form. The `@Valid` annotation is required to have Spring MVC check the validity of `CreateUserFormData` according to the validation annotations that we have used.
- ③ Inject a `BindingResult` instance.
- ④ Check the `bindingResult` instance if there are validation errors. If so, we display the HTML page again.
- ⑤ Add the list of genders again so we can render the radio buttons after a validation error.
- ⑥ Return the name of the Thymeleaf template to render.
- ⑦ Convert the `formData` to domain parameters object `CreateUserParameters` and have the `service` create the user.
- ⑧ Redirect the browser to the list of users.



The `BindingResult` must be *immediately* following the object that is annotated with `@Valid`.

We can now fill in the form:

The screenshot shows a web application window titled "Taming Thymeleaf - Users". The URL in the address bar is "localhost:8080/users/create". On the left, there's a sidebar with icons for Dashboard, Users (selected), Teams, Calendar, Documents, and Reports. The main content area has a title "Create user". It contains a "Gender" section with radio buttons for Male (selected), Female, and Other. Below that are fields for "First name" (Wim), "Last name" (Deblauwe), "Email" (wim.deblauwe@gmail.com), "Phone number" (555-123 456), and "Birthday" (1978-12-02). At the bottom right are "Cancel" and "Save" buttons, with "Save" being green and outlined.

Figure 39. Create user form filled in

After pressing 'Save', we get redirected to `/users`:

NAME	GENDER	BIRTHDAY	EMAIL	
Rosario Barton	MALE	2005-06-24	rosario.barton@yahoo.com	<a href="#">Edit</a>
Raeann Boehm	MALE	1994-02-18	raeann.boehm@gmail.com	<a href="#">Edit</a>
Matha Bradtke	MALE	1980-11-20	matha.bradtke@hotmail.com	<a href="#">Edit</a>
Wim Deblauwe	MALE	1978-12-02	wim.deblauwe@gmail.com	<a href="#">Edit</a>
Casimira DuBuque	FEMALE	1986-09-11	casimira.dubuque@yahoo.com	<a href="#">Edit</a>
Rigoberto Feeney	MALE	1995-06-09	rigoberto.feeney@hotmail.com	<a href="#">Edit</a>
Sau Ferry	MALE	1995-04-15	sau.ferry@yahoo.com	<a href="#">Edit</a>
Shannan Gaylord	MALE	1996-03-18	shannan.gaylord@hotmail.com	<a href="#">Edit</a>
Clair Gislason	MALE	1994-10-04	clair.gislason@hotmail.com	<a href="#">Edit</a>
Ethelene Haag	FEMALE	1981-11-26	ethelene.haag@gmail.com	<a href="#">Edit</a>

Figure 40. Redirection after save

Note how our new user is present in the list.

Now try to save with invalid or missing input. The form remains visible. This is great as we don't get invalid input this way. However, there is no indication of what is wrong.

#### *Cross-site request forgery*

 Thymeleaf has automatic Cross-Site Request Forgery (CSRF) protection when Spring Security is added to the project. Any `<form>` that has a `th:action` will automatically have a hidden `<input>` with a CSRF token. See [Cross Site Request Forgery \(CSRF\) in the Spring Security documentation](#) for more details.

## 11.2. Error messages

To guide our user about what is wrong exactly when there is a validation error, we should add some validation messages.

Thymeleaf has the `#fields.hasErrors` method available in templates that allows checking if there is a validation error on a field or not.

We can apply this to the `firstName` property for example:

```
<div class="sm:col-span-3">
```

```

<label for="firstName" class="block text-sm font-medium text-gray-700">
    th:text="#{user.firstName}"
    First name
</label>
<div class="mt-1 relative rounded-md shadow-sm">
    <input id="firstName"
        type="text"
        th:field="*{firstName}"
        class="shadow-sm block w-full sm:text-sm border-gray-300
rounded-md"
        th:classappend="#{#fields.hasErrors('firstName')?'border-
red-300 focus:border-red-300 focus:ring-red-500':'focus:ring-green-500
focus:border-green-500'}">
        <div th:if="#{#fields.hasErrors('firstName')}"
            class="absolute inset-y-0 right-0 pr-3 flex items-center
pointer-events-none">
            <svg class="h-5 w-5 text-red-500" fill="currentColor"
viewBox="0 0 20 20">
                <path fill-rule="evenodd"
                    d="M18 10a8 8 0 11-16 0 8 8 0 0116 0zm-7 4a1 1 0
11-2 0 1 1 0 012 0zm-1-9a1 1 0 00-1 1v4a1 1 0 102 0V6a1 1 0 00-1-1z"
                    clip-rule="evenodd"/>
            </svg>
        </div>
    </div>
    <p th:if="#{#fields.hasErrors('firstName')"
        class="mt-2 text-sm text-red-600" id="firstName-error">The first
name cannot be
        empty.</p>
</div>

```

The changes:

- An extra `<div>` is added that contains the SVG error icon. By using `th:if="#{#fields.hasErrors('firstName')}`, this tag is only present when there is an actual error on the `firstName` field.
- An extra `<p>` with the error message text.

If we apply this to all the input fields, we get the following error messages if we try to submit an empty form:

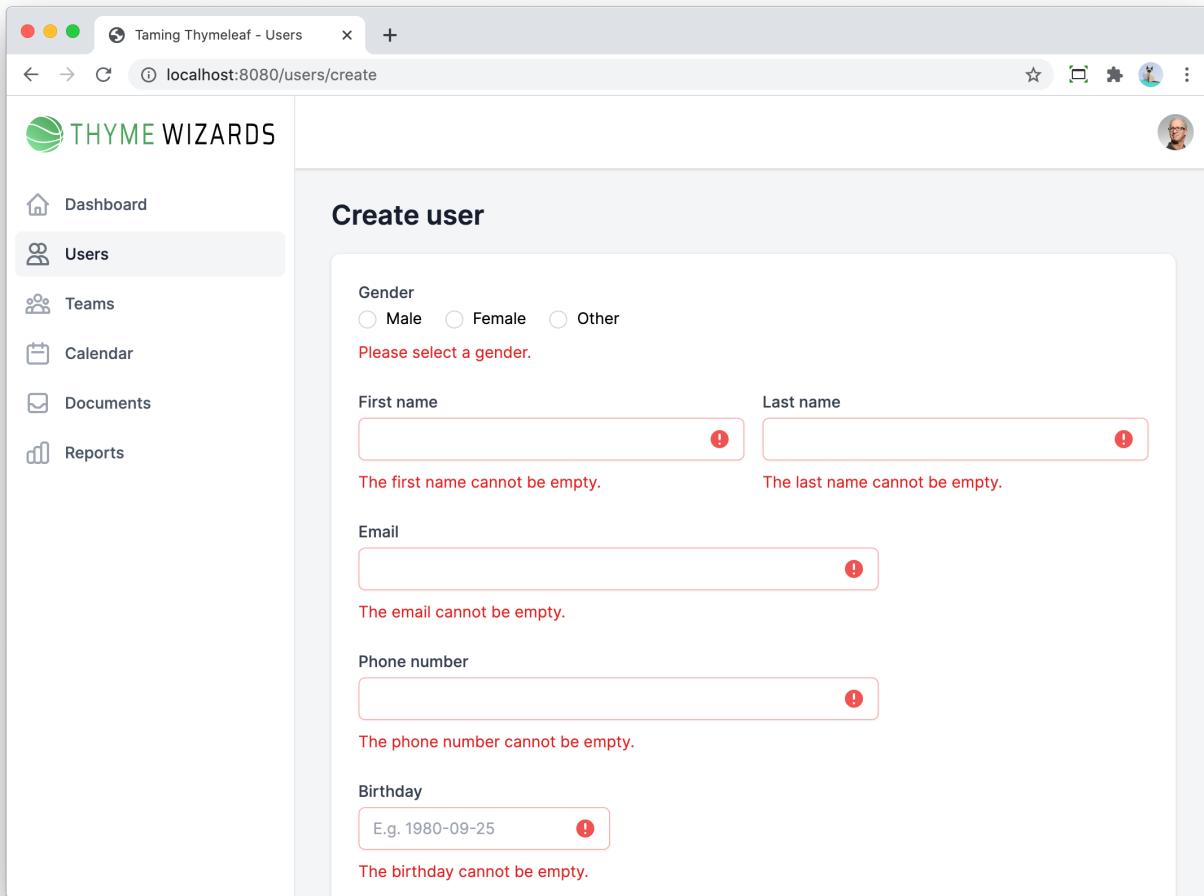


Figure 41. Error messages on empty form submit

However, we have now a hardcoded error message for each field. This means that for fields that can have multiple reasons that the validation fails, we will not be able to show a good error message as it is now.

As an example, take the `birthday` property. These are the validation rules defined in `CreateUserData`:

```
@NotNull
@DateTimeFormat(pattern = "yyyy-MM-dd")
private LocalDate birthday;
```

This means that the value cannot be `null`, but it also has to conform to the pattern.

We can ask for the exact problem(s) on a property using `#fields.errors('property')`. If we apply this to the `birthday` input, we get:

```
<div class="sm:col-span-2">
    <label for="birthday" class="block text-sm font-medium leading-5
text-gray-700">
        th:text="#{user.birthday}">
        Birthday
    </label>
</div>
```

```

</label>
<div class="mt-1 relative rounded-md shadow-sm">
    <input id="birthday"
        type="text"
        th:field="*{birthday}"
        th:placeholder="#{user.birthday.placeholder}"
        class="shadow-sm block w-full sm:text-sm border-gray-300
rounded-md"
        th:classappend="${#fields.hasErrors('birthday')?'border-
red-300 focus:border-red-300 focus:ring-red-500':'focus:ring-green-500
focus:border-green-500'}">
        <div th:if="${#fields.hasErrors('birthday')}">
            class="absolute inset-y-0 right-0 pr-3 flex items-center
pointer-events-none">
                <svg class="h-5 w-5 text-red-500" fill="currentColor"
viewBox="0 0 20 20">
                    <path fill-rule="evenodd"
                        d="M18 10a8 8 0 11-16 0 8 8 0 0116 0zm-7 4a1 1 0
11-2 0 1 1 0 012 0zm-1-9a1 1 0 00-1 1v4a1 1 0 102 0V6a1 1 0 00-1-1z"
                        clip-rule="evenodd"/>
                </svg>
            </div>
        </div>
    <p th:if="${#fields.hasErrors('birthday')}">
        th:text="#{#strings.listJoin(#fields.errors('birthday'), ', ')}"
        class="mt-2 text-sm text-red-600" id="birthday-error">Birthday
validation error message(s).</p>
</div>

```

The most important part is this:

```
th:text="#{#strings.listJoin(#fields.errors('birthday'), ', ')}"
```

We first get the list of errors for the `birthday` property (Technically a `List<String>`) and concatenate those to form a comma separated list.

Applying this to all the fields, we get this result:

The screenshot shows a web application window titled "Taming Thymeleaf - Users" at the URL "localhost:8080/users/create". The left sidebar has a "Users" item selected. The main content area is titled "Create user". It contains a "Gender" section with three radio buttons: "Male", "Female", and "Other". Below it is a note "must not be null". The "First name" field is empty and has a red border with an exclamation mark, with the error message "must not be blank". The "Last name" field is empty and has a red border with an exclamation mark, with the error message "must not be blank". The "Email" field is empty and has a red border with an exclamation mark, with the error message "must not be blank". The "Phone number" field is empty and has a red border with an exclamation mark, with the error message "must match \"[0-9.\\-() x/+]+\"". The "Birthday" field contains "E.g. 1980-09-25" and has a red border with an exclamation mark, with the error message "must not be null". A user profile picture is visible in the top right corner.

Figure 42. Default error messages

If we enter something invalid for birthday:

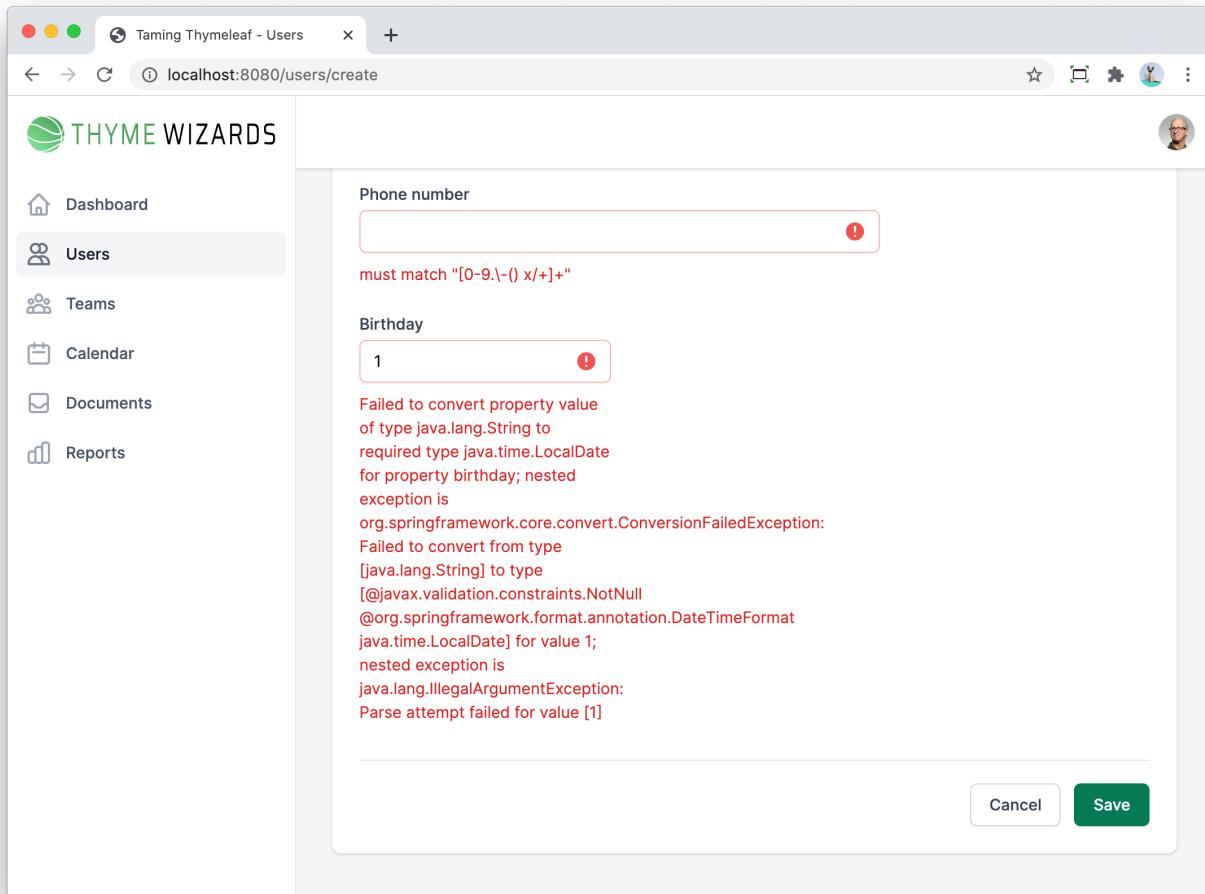


Figure 43. Default error message if birthday has wrong format

We have now very *technically correct* error messages, but not so user friendly. Let's fix this in the next part.

## 11.3. Custom error messages

Using the [Internationalization](#) support we have set up before, we can change the error messages to something user-friendly, and have it translated for each language.

We can change the error message for all validation errors of a certain type (E.g. all `@NotNull` violations), or we can precisely change the message for a single property on a single model attribute.

This is an example of the 4 options:

```
NotBlank=The property '{0}' should not be blank.  
NotBlank.email=Please enter the email address.  
NotBlank.java.lang.String=The string should not be empty.  
NotBlank.user.birthday=Please enter the birthday of the user.
```

- **NotBlank**: This will affect *all* fields from *all* objects that use the `@NotNull` validation annotation. The `{0}` is replaced with the name of the field. An example error message would be: `The property 'First name' should not be blank.`

- **NotBlank.email**: This will affect all fields named `email` from *all* objects that use the `@NotBlank` validation annotation.
- **NotBlank.java.lang.String**: This will affect *all* fields from *all* objects that use the `@NotBlank` validation annotation *and* are of type `java.lang.String`.
- **NotBlank.user.birthday**: This message will only be shown for the `birthday` field of a model attribute named `user` that is annotated with `@NotBlank`.

If we apply this to our example, we can add the following to `src/main/resources/i18n/messages.properties`:

```
NotBlank.user.firstName=Please enter the first name.  
NotBlank.user.lastName=Please enter the last name.  
NotNull.user.gender=Please select the gender.  
NotBlank.user.email=Please enter the email address.  
NotNull.user.birthday=Please enter the users' birthday.  
typeMismatch.user.birthday=Please use the correct format for the  
birthday: YYYY-MM-DD  
NotNull.user.phoneNumber=Please enter the phone number.  
Pattern.user.phoneNumber=Please enter a valid phone number.
```

We now have proper user-friendly error messages.

The screenshot shows a web application window titled "Taming Thymeleaf - Users". The URL in the address bar is "localhost:8080/users/create". The left sidebar contains navigation links: Dashboard, Users (which is selected), Teams, Calendar, Documents, and Reports. The main content area has a header "THYME WIZARDS" and a user profile picture. The form for creating a new user has the following fields:

- Gender:** Radio buttons for Male, Female, and Other. A message below says "Please select the gender."
- First name:** An input field with a red border and an exclamation mark icon. A message below says "Please enter the first name."
- Last name:** An input field with a red border and an exclamation mark icon. A message below says "Please enter the last name."
- Email:** An input field with a red border and an exclamation mark icon. A message below says "Please enter the email address."
- Phone number:** An input field with a red border and an exclamation mark icon. A message below says "Please enter a valid phone number."
- Birthday:** An input field containing the number "1" with a red border and an exclamation mark icon. A message below says "Please use the correct format for the birthday: YYYY-MM-DD"

Figure 44. Custom error messages

They can also be translated by adding the same translation keys to the other `messages` files (E.g. `messages_nl.properties`)

The screenshot shows a web application titled "THYME WIZARDS" with a sidebar containing links for Dashboard, Gebruikers (selected), Teams, Kalender, Documenten, and Rapporten. The main content area is a form for creating a new user. The form fields are:

- Geslacht:** Radio buttons for Man, Vrouw, and Ander. An error message "Gelieve het geslacht te kiezen." is displayed below.
- Voornaam:** An input field with an error message "Gelieve de voornaam in te vullen." Below it is another error message "Gelieve de familie naam in te vullen."
- Familie naam:** An input field with an error message "Gelieve de familie naam in te vullen."
- Email:** An input field with an error message "Gelieve het email adres in te vullen."
- Telefoon:** An input field with an error message "Gelieve een geldig telefoon nummer in te vullen."
- Verjaardag:** An input field with an error message "Gelieve het juiste formaat voor de verjaardag te gebruiken: JJJJ-MM-DD".

Figure 45. Custom error messages translated in Dutch

Some validation annotations allow to pass extra arguments to the translations. For example `Size`:

```
@NotBlank
@Size(min = 2, max = 200)
private String firstName;
```

We can use those arguments in the custom validation message:

```
Size.user.firstName=The first name should be between {2} and {1}
characters.
```

If the name is too long, the error message will print: `The first name should be between 2 and 200 characters.`

## 11.4. Custom validator

We have been using the "built-in constraints" from the `javax.validation.constraints` package until now. When those are not enough, you can create your own custom validation. Such a validation can validate a single field, or the whole class. Check out the [Custom Validator to check if a String contains](#)

[XML](#) blog post for an example of a custom validation on a single field.

We will create a custom validator that validates the whole object. For example, it would be good to validate if there is already a known user with the given email address since the email address of each user should be unique.

We start by creating our own annotation:

```
package com.tamingthymeleaf.application.user.web;

import javax.validation.Constraint;
import javax.validation.Payload;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(ElementType.TYPE) ①
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = NotExistingUserValidator.class) ②
public @interface NotExistingUser {
    String message() default "{UserAlreadyExisting}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};
}
```

① Indicates that the custom annotation can be applied only to a class.

② Reference the validator that will be responsible for doing the actual validation via the `@Constraint` annotation.

Next, we create the validator itself:

```
package com.tamingthymeleaf.application.user.web;

import com.tamingthymeleaf.application.user.Email;
import com.tamingthymeleaf.application.user.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.util.StringUtils;

import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;

public class NotExistingUserValidator implements ConstraintValidator<NotExistingUser, CreateUserData> {
```

```

private final UserService userService;

@Autowired
public NotExistingUserValidator(UserService userService) { ①
    this.userService = userService;
}

public void initialize(NotExistingUser constraint) { ②
    // intentionally empty
}

public boolean isValid(CreateUserData formData,
ConstraintValidatorContext context) {
    if (!StringUtils.isEmpty(formData.getEmail())
        && userService.userWithEmailExists(new Email(formData
.getEmail()))) { ③
        context.disableDefaultConstraintViolation(); ④
        context.buildConstraintViolationWithTemplate
("'{UserAlreadyExisting}'") ⑤
            .addPropertyNode("email") ⑥
            .addConstraintViolation(); ⑦

        return false; ⑧
    }

    return true;
}
}

```

- ① Inject the `UserService` Spring component so we can check if there is already an existing user with the given email address.
- ② `initialize` method is not needed in this example. This is useful when your custom annotation has extra parameters you want to read out.
- ③ Check if there is an existing user via `userService`. Note how we also need to check if the `email` is not empty first. This is because our validator currently runs *before* the field validations `@NotBlank` and `@Email`. The section `Validation groups and order` further in the book will show how we can make those run first and avoid that check.
- ④ Disables the default constraint violation: Because our validator is used with an annotation on class level, Spring will register a constraint violation at the "global" level by default. We don't want this in this example, as we will register it on the `email` property level. If we would not call this method, there would be the same error message twice. Once at the global level and once for the `email` property.
- ⑤ Create a `ConstraintViolationBuilder`. By using the `{...}` syntax, the actual message can come from the `messages.properties` file (and can be translated).

- ⑥ By calling `addPropertyNode("email")`, the violation will be tied to the `email` field.
- ⑦ Finish the builder configuration and add the constraint violation.
- ⑧ Return false to indicate that the object is not valid.

We use the `{...}` annotation to be able to translate the error messages. To be able to read those translations from `messages.properties`, we need to configure it. We do this by creating our own instance

`org.springframework.validation.beanvalidation.LocalValidatorFactoryBean` in  
`TamingThymeleafApplicationConfiguration`:

```
package com.tamingthymeleaf.application;

import io.github.wimdeblauwe.jpearl.InMemoryUniqueIdGenerator;
import io.github.wimdeblauwe.jpearl.UniqueIdGenerator;
import org.springframework.context.MessageSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.validation.beanvalidation.LocalValidatorFactoryBean;
import
org.thymeleaf.spring5.templateresolver.SpringResourceTemplateResolver;
import org.thymeleaf.templateresolver.ITemplateResolver;

import java.util.UUID;

@Configuration
public class TamingThymeleafApplicationConfiguration {

    @Bean
    public ITemplateResolver svgTemplateResolver() {
        SpringResourceTemplateResolver resolver = new
SpringResourceTemplateResolver();
        resolver.setPrefix("classpath:/templates/svg/");
        resolver.setSuffix(".svg");
        resolver.setTemplateMode("XML");

        return resolver;
    }

    @Bean
    public UniqueIdGenerator<UUID> uniqueIdGenerator() {
        return new InMemoryUniqueIdGenerator();
    }

    @Bean
```

```

public LocalValidatorFactoryBean localValidatorFactoryBean
(MessageSource messageSource) { ①
    LocalValidatorFactoryBean bean = new
LocalValidatorFactoryBean();
    bean.setValidationMessageSource(messageSource); ②
    return bean;
}
}

```

① Declare a new bean and have Spring inject the `MessageSource` instance.

② Inject the `MessageSource` into the `LocalValidatorFactoryBean`.

We can now add the validation message key to `messages.properties`:

```
UserAlreadyExisting=There is already a user with the given email
address.
```

Everything is now in place to use the new `@NotExistingUser` annotation on `CreateUserData`:

`com.tamingthymeleaf.application.user.web.CreateUserData`

```

@NotExistingUser
public class CreateUserData {
    ...
}

```

After also updating `UserService`, `UserService` and `UserRepository` with the supporting code to check if there is already a user with the given email present, we get the following result:

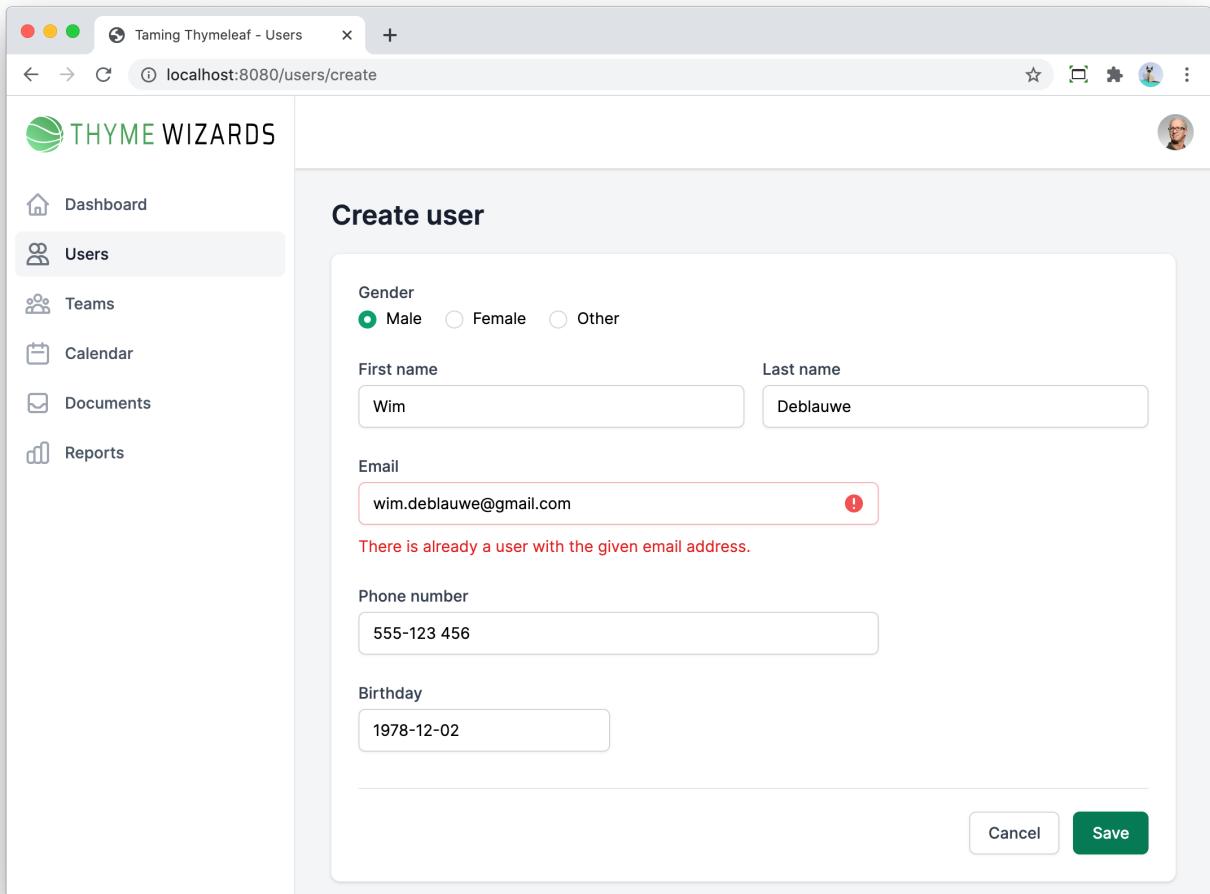


Figure 46. Custom validator checks if the user is known already



The `addPropertyNode("property")` is optional. When not used, the error is tied to the object and not to a particular field.

## 11.5. Errors summary

Sometimes, you will want to add an errors summary when validation fails. This is usually put at the top of the form.

To implement this with Thymeleaf, we create a fragment `fielderrors` to iterate over all errors and display them in a bullet list.

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

<div th:fragment="fielderrors"
      class="rounded-md bg-red-50 p-4 mb-4"
      th:if="#{#fields.hasErrors()}"> ①
    <div class="flex">
      <div class="flex-shrink-0">
```

```

        <svg class="h-5 w-5 text-red-400"
xmlns="http://www.w3.org/2000/svg" viewBox="0 0 20 20"
fill="currentColor" aria-hidden="true">
    <path fill-rule="evenodd" d="M10 18a8 8 0 100-16 8 8 0
000 16zM8.707 7.293a1 1 0 00-1.414 1.414L8.586 10l-1.293 1.293a1 1 0
101.414 1.414L10 11.414l1.293 1.293a1 1 0 001.414-1.414L11.414 10l1.293-
1.293a1 1 0 00-1.414-1.414L10 8.586 8.707 7.293z" clip-rule="evenodd" />
</svg>
</div>
<div class="ml-3">
    <h3 class="text-sm font-medium text-red-800"

th:text="#{error.messages.summary.title(${#fields.errors().size()})}">
②
    There were 2 errors with your submission
</h3>
<div class="mt-2 text-sm text-red-700">
    <ul class="list-disc pl-5 space-y-1">
        <li th:each="err,iter : ${#fields.errors()}">
th:text="${err}"></li> ③
        </ul>
    </div>
    </div>
</div>
</html>

```

- ① Only have the `<div>` displayed if there are actually errors.
- ② Set the summary title and pass in the number of errors found.
- ③ Create `<li>` tags for each error with the error message.

Now add the fragment to `users/edit.html`:

```

<form id="user-form"
    th:object="${user}"
    th:action="@{/users/create}"
    method="post">
    <div>
        <div th:replace="fragments/forms :: fielderrors"></div>
        <div class="grid grid-cols-1 row-gap-6 col-gap-4 sm:grid-cols-6">
...

```

This gives the following result:

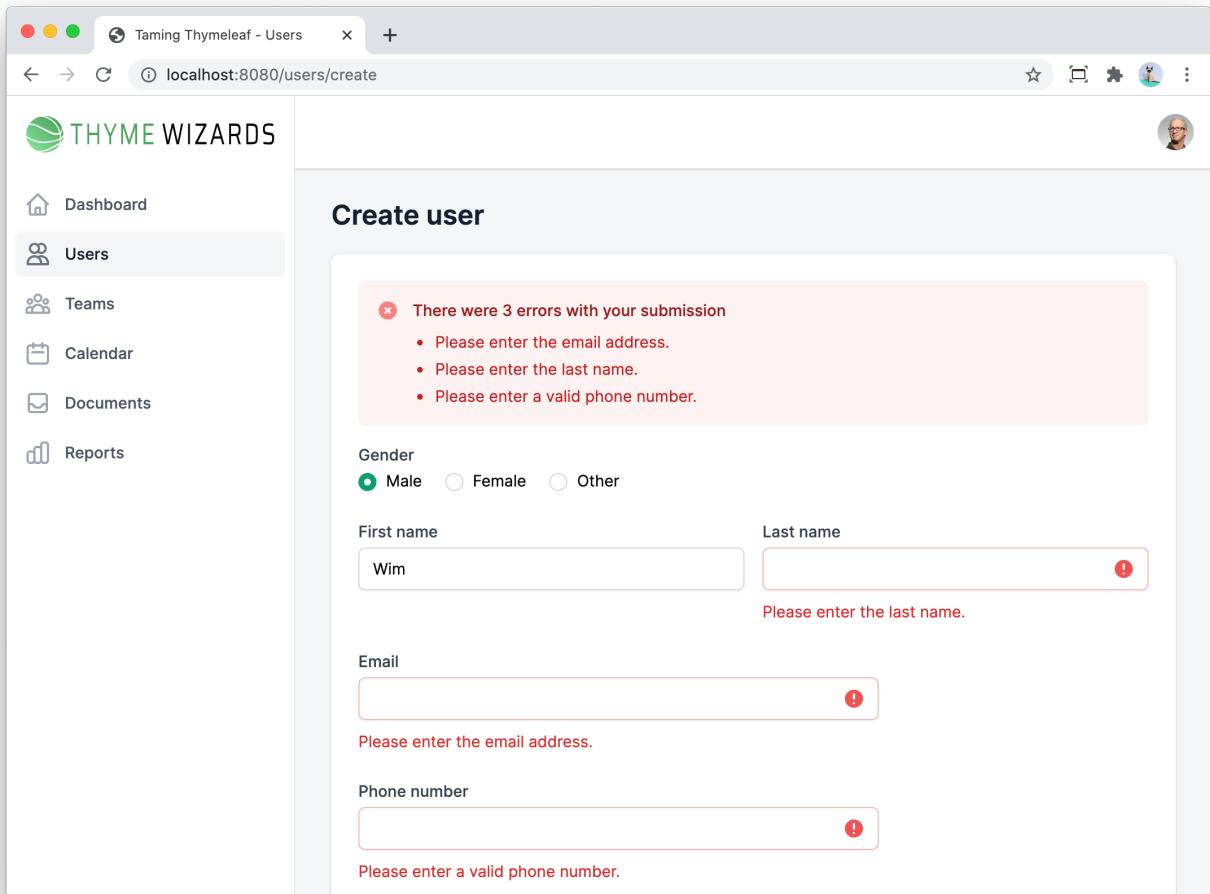


Figure 47. Errors summary at the top of the form

If you test this a bit, you might notice that the order of the error messages changes (almost) every time. If this is undesired, you can sort the messages based on the name of the corresponding field for example.

To do that, use `#fields.detailedErrors()` (Which returns `DetailedError` instances instead of `Strings`) and a custom comparator:

```
<ul class="list-disc pl-5">
    <li th:each="detailedError,iter :
${#lists.sort(#fields.detailedErrors(), new
com.tamingthymeleaf.application.infrastructure.web.Detailed
ErrorComparator())}" th:text=
"${detailedError.message}"></li>
</ul>
```



The sources of `DetailedErrorComparator`:

```
package com.tamingthymeleaf.application.infrastructure.web;

import org.thymeleaf.spring5.util.DetailedError;
```

```

import java.util.Comparator;

public class DetailedErrorComparator implements Comparator<DetailedError> {

    @Override
    public int compare(DetailedError o1, DetailedError o2) {
        return o1.getFieldName().compareTo(o2.
getFieldName());
    }
}

```

## 11.6. Validation groups and order

In the [Custom validator](#) section, we had to ensure that the email was not empty in our custom validator, due to the undefined order of the validations.

We can influence the processing order of the validations by using validation groups.

Suppose we add some extra validations on [CreateUserFormData](#), for example:

```

@NotExistingUser
public class CreateUserFormData {
    @NotBlank
    @Size(min = 1, max = 200)
    private String firstName;
    @NotBlank
    @Size(min = 1, max = 200)
    private String lastName;
    @NotNull
    private Gender gender;
    @NotBlank
    @Email
    private String email;
    @NotNull
    @DateTimeFormat(pattern = "yyyy-MM-dd")
    private LocalDate birthday;
    @NotBlank
    @Pattern(regexp = "[0-9.\\-() x/+]+")
    private String phoneNumber;
}

```

Without using validation groups, all validations are triggered at the same time, resulting in 2

problems:

- There is no defined order, so our `@NotExistingUser` annotation cannot be sure that the `email` is not blank and a valid email address.
- The user is shown 2 error messages for empty input fields. Once for `@NotBlank` and once for `@Size`.

The screenshot shows a web application titled "Taming Thymeleaf - Users" at the URL "localhost:8080/users/create". The left sidebar has a "Users" tab selected, which is highlighted in grey. The main content area displays a user creation form with the following fields and errors:

- Gender:** Radio buttons for Male, Female, and Other. An error message "Please select the gender." is displayed below the buttons.
- First name:** An empty input field with a red border and an exclamation mark icon. Two error messages are shown: "The first name cannot exceed 200 characters., Please enter the first name." and "Please enter the last name., size must be between 1 and 200".
- Last name:** An empty input field with a red border and an exclamation mark icon. The error message "Please enter the last name., size must be between 1 and 200" is displayed.
- Email:** An empty input field with a red border and an exclamation mark icon. The error message "Please enter the email address." is displayed.
- Phone number:** An empty input field with a red border and an exclamation mark icon. The error message "Please enter a valid phone number., Please enter the phone number." is displayed.
- Birthday:** An input field containing "E.g. 1980-09-25" with a red border and an exclamation mark icon. The error message "Please enter the users birthday." is displayed.

At the bottom right of the form are two buttons: "Cancel" and "Save".

Figure 48. Without validation groups, multiple errors per input field are shown

We first need to define a marker interface for each validation "phase":

```
package com.tamingthymeleaf.application.infrastructure.validation;

public interface ValidationGroupOne { }
```

```
package com.tamingthymeleaf.application.infrastructure.validation;

public interface ValidationGroupTwo { }
```

Next, we define the validation order in another marker interface that is annotated with [@GroupSequence](#):

```
package com.tamingthymeleaf.application.infrastructure.validation;

import javax.validation.GroupSequence;
import javax.validation.groups.Default;

@GroupSequence({Default.class, ValidationGroupOne.class,
ValidationGroupTwo.class})
public interface ValidationGroupSequence {
}
```

The order of the arguments of [@GroupSequence](#) defines the order of validation processing.

For each of the validation annotations, we assign them to the default group, or one of our new groups:

```
package com.tamingthymeleaf.application.user.web;

import
com.tamingthymeleaf.application.infrastructure.validation.ValidationGrou
pOne;
import
com.tamingthymeleaf.application.infrastructure.validation.ValidationGrou
pTwo;
import com.tamingthymeleaf.application.user.CreateUserParameters;
import com.tamingthymeleaf.application.user.Gender;
import com.tamingthymeleaf.application.user.PhoneNumber;
import com.tamingthymeleaf.application.user.UserName;
import org.springframework.format.annotation.DateTimeFormat;

import javax.validation.constraints.*;
import java.time.LocalDate;

@NotExistingUser(groups = ValidationGroupTwo.class)
public class CreateUserData {
    @NotBlank
    @Size(min = 1, max = 200, groups = ValidationGroupOne.class)
    private String firstName;
    @NotBlank
    @Size(min = 1, max = 200, groups = ValidationGroupOne.class)
    private String lastName;
    @NotNull
    private Gender gender;
```

```

    @NotBlank
    @Email(groups = ValidationGroupOne.class)
    private String email;
    @NotNull
    @DateTimeFormat(pattern = "yyyy-MM-dd")
    private LocalDate birthday;
    @NotBlank
    @Pattern(regexp = "[0-9.\\-() x/+]+", groups = ValidationGroupOne
.class)
    private String phoneNumber;

    // Getters and setters omitted

}

```

Finally, we need to replace `javax.validation.Valid` with `org.springframework.validation.annotation.Validated` on the controller method and reference our `CreateUserValidationGroupSequence`:

```

    @PostMapping("/create")
    public String doCreateUser(@Validated(ValidationGroupSequence.class)
    @ModelAttribute("user") CreateUserData formData,
                               BindingResult bindingResult, Model model)
    {
        if (bindingResult.hasErrors()) {
            model.addAttribute("genders", List.of(Gender.MALE, Gender
.FEMALE, Gender.OTHER));
            return "users/edit";
        }

        service.createUser(formData.toParameters());
        return "redirect:/users";
    }

```

This results in the following behaviour:

- All annotations that do not have the `groups` variable set are part of the default group and are evaluated first. If they fail, the validations from the other groups (`ValidationGroupOne` and `ValidationGroupTwo`) are not evaluated.
- If everything is ok for the default group, all validations from `ValidationGroupOne` are evaluated.
- If everything is ok for `ValidationGroupOne`, then `ValidationGroupTwo` is evaluated.

By putting our custom `@NotExistingUser` in `ValidationGroupTwo`, it is executed after the other validation in the default group and in group `ValidationGroupOne`. The `email` field has `@NotBlank`

and `@Email(groups = ValidationGroupOne.class)` annotations, so those are executed before our `NotExistingUserValidator` kicks in. Due to that, we can remove the `!StringUtils.isEmpty` check from `NotExistingUserValidator`:

```
public boolean isValid(CreateUserData formData,
ConstraintValidatorContext context) {
    if (userService.userWithEmailExists(new Email(formData.
getEmail()))) {
        context.disableDefaultConstraintViolation();
        context.buildConstraintViolationWithTemplate
("UserAlreadyExisting")
            .addPropertyNode("email")
            .addConstraintViolation();

        return false;
    }

    return true;
}
```

## 11.7. Summary

In this chapter, you learned:

- How to create an HTML form and submit the data from it.
- How to validate user input on the server-side.
- How to display custom error messages.
- How to write your own custom validator.

# Chapter 12. Data editing

So far, we have been working on an HTML form to add a new user. We can adjust that `edit.html` template a bit to also allow editing of a user, next to creation of the user.

## 12.1. Add user button

We will start by adding an 'Add user' button to avoid having to enter the `http://localhost:8080/users/create` URL manually:

NAME	GENDER	BIRTHDAY	EMAIL	
Rosario Barton	MALE	2005-06-24	rosario.barton@yahoo.com	<a href="#">Edit</a>
Raeann Boehm	MALE	1994-02-18	raeann.boehm@gmail.com	<a href="#">Edit</a>
Matha Bradtke	MALE	1980-11-20	matha.bradtke@hotmail.com	<a href="#">Edit</a>
Wim Deblauwe	MALE	1978-12-02	wim.deblauwe@gmail.com	<a href="#">Edit</a>
Casimira DuBuque	FEMALE	1986-09-11	casimira.dubuque@yahoo.com	<a href="#">Edit</a>
Rigoberto Feeney	MALE	1995-06-09	rigoberto.feeney@hotmail.com	<a href="#">Edit</a>
Sau Ferry	MALE	1995-04-15	sau.ferry@yahoo.com	<a href="#">Edit</a>
Shannan Gaylord	MALE	1996-03-18	shannan.gaylord@hotmail.com	<a href="#">Edit</a>
Clair Gislason	MALE	1994-10-04	clair.gislason@hotmail.com	<a href="#">Edit</a>
Ethelene Haag	FEMALE	1981-11-26	ethelene.haag@gmail.com	<a href="#">Edit</a>

Figure 49. Add user button in top right corner

Since we will have multiple pages with such a button in the top right corner, we will create a fragment for it:

`src/main/resources/templates/fragments/titles.html`

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org"
      lang="en">
<div th:fragment="title-with-button(title, buttonIcon, buttonText,
buttonLink)">
```

```

    class="max-w-7xl mx-auto px-4 sm:px-6 md:px-8" > ①
<div class="flex justify-between">
    <h1 class="text-2xl font-semibold text-gray-900"
        th:text="${title}">Title</h1> ②
    <span class="inline-flex rounded-md shadow-sm">
<a th:href="${buttonLink}"
    class="inline-flex items-center px-4 py-2 border border-
transparent shadow-sm text-base font-medium rounded-md text-white bg-
green-600 hover:bg-green-700 focus:outline-none focus:ring-2 focus:ring-
offset-2 focus:ring-green-500"> ③
    <svg th:replace="__${buttonIcon}__"></svg> ④
    [[${buttonText}]] ⑤
</a>
    </span>
</div>
</div>
</html>

```

① Define the fragment name with the 4 arguments:

- The title (that is shown on the left side)
- The button icon
- The button text
- The button link

② Use the title variable for the `<h1>` tag.

③ Use the link variable for the `th:href`.

④ Use the name of the button icon with [Thymeleaf preprocessing](#) so the name is replaced first, before the `th:replace` kicks in to get the actual SVG icon.

⑤ Use the `buttonText` variable with [Expression inlining](#).



Do not use `th:text` on `<button>` as that would hide the child `<svg>` tag.

Add the SVG icon to `templates/svg/user-add.svg`:

```

<svg viewBox="0 0 20 20" fill="currentColor" class="-ml-1 mr-2 w-5 h-5">
    <path
        d="M8 9a3 3 0 100-6 3 3 0 000 6zM8 11a6 6 0 016 6H2a6 6 0 016-
6zM16 7a1 1 0 10-2 0v1h-1a1 1 0 100 2h1v1a1 1 0 102 0v-1h1a1 1 0 100-2h-
1V7z"></path>
</svg>

```

We can now use the fragment in `users/list.html`:

```
<div th:replace="fragments/titles :: title-with-
button(${users.title}, 'user-add', ${users.add},
@{/users/create})"></div>
```

## 12.2. Edit user data

To support editing of our `User` entities, we will need some Java code changes.

First of all, we will use `io.github.wimdeblauwe.jpearl.AbstractVersionedEntity` as superclass of `User` instead of `io.github.wimdeblauwe.jpearl.AbstractEntity`. This class adds a `version` field to the entity, which will allow using Optimistic Locking.

Most important here is that this will guard the user from concurrent updates (either by another user, or by himself in another tab for example).

We need to update our SQL creation script to have the new `version` field:

`src/main/resources/db/migration/V1.0_init.sql`

```
CREATE TABLE tt_user
(
    id          UUID      NOT NULL,
    version     BIGINT    NOT NULL, ①
    first_name  VARCHAR   NOT NULL,
    last_name   VARCHAR   NOT NULL,
    gender      VARCHAR   NOT NULL,
    birthday    DATE      NOT NULL,
    email       VARCHAR   NOT NULL,
    phone_number VARCHAR  NOT NULL,
    PRIMARY KEY (id)
);
```

① `version` field added

Because we now edit the SQL script, we will have to drop the database tables and have Flyway create them again:

```
DROP SCHEMA public CASCADE;
CREATE SCHEMA public;
```



If you want to avoid that, create a 2nd migration script with an `ALTER TABLE` statement to add the `version` column.

I like to edit the initial script as long as I am developing to avoid having many alterations that will really serve no purpose once the software has the first release. Once the first release is done, it is important to *not* alter this file anymore and create

a new one for the next release.

Next, create a new `UserService` method to allow editing user properties:

`com.tamingthymeleaf.application.user.UserService`

```
User editUser(UserId userId, EditUserParameters parameters);
```

Since we will allow to edit all parameters used at creation time, we can extend from `CreateUserParameters` and add the `version` field:

```
package com.tamingthymeleaf.application.user;

import java.time.LocalDate;

public class EditUserParameters extends CreateUserParameters {
    private final long version;

    public EditUserParameters(long version, UserName userName, Gender
gender, LocalDate birthday, Email email, PhoneNumber phoneNumber) {
        super(userName, gender, birthday, email, phoneNumber);
        this.version = version;
    }

    public long getVersion() {
        return version;
    }

}
```

Now update `UserServiceImpl`:

`com.tamingthymeleaf.application.user.UserServiceImpl`

```
@Override
public User editUser(UserId userId, EditUserParameters parameters) {
    User user = repository.findById(userId)
        .orElseThrow(() -> new
UserNotFoundException(userId)); ①

    if (parameters.getVersion() != user.getVersion()) { ②
        throw new ObjectOptimisticLockingFailureException(User.
class, user.getId().asString());
    }

    parameters.update(user); ③
```

```

    return user;
}

```

- ① Get the user for the given `UserId` from the database. If there is no matching user, throw a `UserNotFoundException`.
- ② Check if the version that is passed from the `parameters` (which will come from the HTML form) is equal to the current version in the database.
- ③ Have the `parameters` object update the properties of the `user`.



There is no need to call `repository.save(user)`. This is done automatically by JPA/Hibernate.

The `UserNotFoundException` is a custom exception that looks like this:

```

package com.tamingthymeleaf.application.user;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(HttpStatus.NOT_FOUND)
public class UserNotFoundException extends RuntimeException {
    public UserNotFoundException(UserId userId) {
        super(String.format("User with id %s not found", userId.
asString()));
    }
}

```

The `update` method in `EditUserParameters` should be added:

`com.tamingthymeleaf.application.user.EditUserParameters`

```

public void update(User user) {
    user.setUserName(getUserName());
    user.setGender(getGender());
    user.setBirthday(getBirthday());
    user.setEmail(getEmail());
    user.setPhoneNumber(getPhoneNumber());
}

```

We can now focus on the web part of updating the user. We again need an object that represents the form data: `EditUserFormData`

```

package com.tamingthymeleaf.application.user.web;

import com.tamingthymeleaf.application.user.*;

```

```
public class EditUserData extends CreateUserData { ①
    private String id; ②
    private long version; ③

    public static EditUserData fromUser(User user) { ④
        EditUserData result = new EditUserData();
        result.setId(user.getId().asString());
        result.setVersion(user.getVersion());
        result.setFirstName(user.getUserName().getFirstName());
        result.setLastName(user.getUserName().getLastName());
        result.setGender(user.getGender());
        result.setBirthday(user.getBirthday());
        result.setEmail(user.getEmail().asString());
        result.setPhoneNumber(user.getPhoneNumber().asString());

        return result;
    }

    public EditUserParameters toParameters() { ⑤
        return new EditUserParameters(version,
            new UserName(getFirstName(),
getLastName()),
            getGender(),
            getBirthday(),
            new Email(getEmail()),
            new PhoneNumber(
getPhoneNumber()));
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public long getVersion() {
        return version;
    }

    public void setVersion(long version) {
        this.version = version;
    }
}
```

```
}
```

- ① Extend from `CreateUserFormData` since we allow to edit all the same fields.
- ② Add the `id` field to reference the `UserId`
- ③ Add the `version` field to keep track of the version of the `User` that we edit.
- ④ A `static` factory method to create a filled in `EditUserFormData` instance given a `User`.
- ⑤ A conversion method to convert the form data to the rich value object `EditUserParameters`.

We again have to implement the GET-POST-REDIRECT cycle as we did for `user creation`. This is the GET mapping in `UserController`:

```
com.tamingthymeleaf.application.user.web.UserController
```

```
@GetMapping("/{id}") ①
public String editUserForm(@PathVariable("id") UserId userId, ②
                           Model model) {
    User user = service.getUser(userId)
        .orElseThrow(() -> new UserNotFoundException
(userId)); ③
    model.addAttribute("user", EditUserFormData.fromUser(user)); ④
    model.addAttribute("genders", List.of(Gender.MALE, Gender.
FEMALE, Gender.OTHER));
    model.addAttribute("editMode",EditMode.UPDATE); ⑤
    return "users/edit"; ⑥
}
```

- ① The URL for editing will be `/users/{id}` where `{id}` is the textual representation of the `UserId`.
  - ② Map the `{id}` part of the URL to the `UserId` variable via the `@PathVariable` annotation.
  - ③ Get the `User` from the database so we can display the current user values in the form.
  - ④ Create an `EditUserFormData` instance which we will bind to the form fields in the HTML page.
  - ⑤ Since we will share the same `users/edit.html` template for user creation and user editing, we need to know in what "mode" we are currently working. For that reason, a model attribute `EditMode` is added to the model. In the `createUserForm` method, we need to add `model.addAttribute("editMode",EditMode.CREATE)`;
- These are the sources for the `EditMode` enum:

```
package com.tamingthymeleaf.application.infrastructure.web;

public enum EditMode {
    CREATE,
    UPDATE
}
```

We need to extend `UserService` and `UserServiceImpl` to be able to get a `User` given a certain

`UserId` for the controller:

`com.tamingthymeleaf.application.user.UserServiceImpl`

```
@Override
public Optional<User> getUser(UserId userId) {
    return repository.findById(userId);
}
```

To be able to use the `UserId` as `@PathVariable`, we have to tell Spring how to convert from the String that the URL is, to our value object. This is done by implementing an `org.springframework.core.convert.converter.Converter`:

```
package com.tamingthymeleaf.application.user.web;

import com.tamingthymeleaf.application.user.UserId;
import org.springframework.core.convert.converter.Converter;
import org.springframework.stereotype.Component;

import java.util.UUID;

@Component
public class StringToUserIdConverter implements Converter<String,
UserId> { ①
    @Override
    public UserId convert(String s) {
        return new UserId(UUID.fromString(s)); ②
    }
}
```

① Use the proper generics to have conversion from `String` to `UserId`

② Implement the actual conversion.

We are getting close to already showing the current values of a user. We just need to make the 'Edit' links in the list of users point to the good URL:

`src/main/resources/templates/users/list.html`

```
<tr class="bg-white" th:each="user : ${users}">
    <td th:replace="fragments/table ::"
data(contents=${user.userName.fullName},primary=true)"></td>
    <td th:replace="fragments/table ::"
data(contents=${user.gender},hideOnMobile=true)"></td>
    <td th:replace="fragments/table ::"
data(contents=${user.birthday},hideOnMobile=true)"></td>
    <td th:replace="fragments/table ::"
```

```
data(contents=${user.email.asString()}, hideOnMobile=true)"></td>
<td th:replace="fragments/table :: dataWithLink('Edit', @{'/users/' +
${user.id.asString()}})"></td> ①
</tr>
```

① Get the string representation of the `UserId` and build the good URL

Starting the application shows the correct links on in the list of users:

NAME	GENDER	BIRTHDAY	EMAIL	
Lupe Buckridge	FEMALE	1994-01-15	lupe.buckridge@gmail.com	<a href="#">Edit</a>
Francine Connelly	FEMALE	1988-07-29	francine.connelly@gmail.com	<a href="#">Edit</a>
Lynn Daniel	FEMALE	2007-05-21	lynn.daniel@yahoo.com	<a href="#">Edit</a>
Gordon Farrell	FEMALE	1981-11-13	gordon.farrell@yahoo.com	<a href="#">Edit</a>
Kandis Gerlach	MALE	1999-04-18	kandis.gerlach@gmail.com	<a href="#">Edit</a>
Efrain Glover	FEMALE	1994-08-14	efrain.glover@gmail.com	<a href="#">Edit</a>
Kelley Grimes	MALE	2008-04-24	kelley.grimes@yahoo.com	<a href="#">Edit</a>
Irving Hill	MALE	1989-05-31	irving.hill@yahoo.com	<a href="#">Edit</a>
Cornell Jacobson	MALE	1987-01-17	cornell.jacobson@hotmail.com	<a href="#">Edit</a>
Shelby Johnston	MALE	1982-03-11	shelby.johnston@hotmail.com	<a href="#">Edit</a>

Figure 50. Hovering over 'Edit' shows the link in the bottom left corner

Clicking on the 'Edit' link shows the details of the selected user:

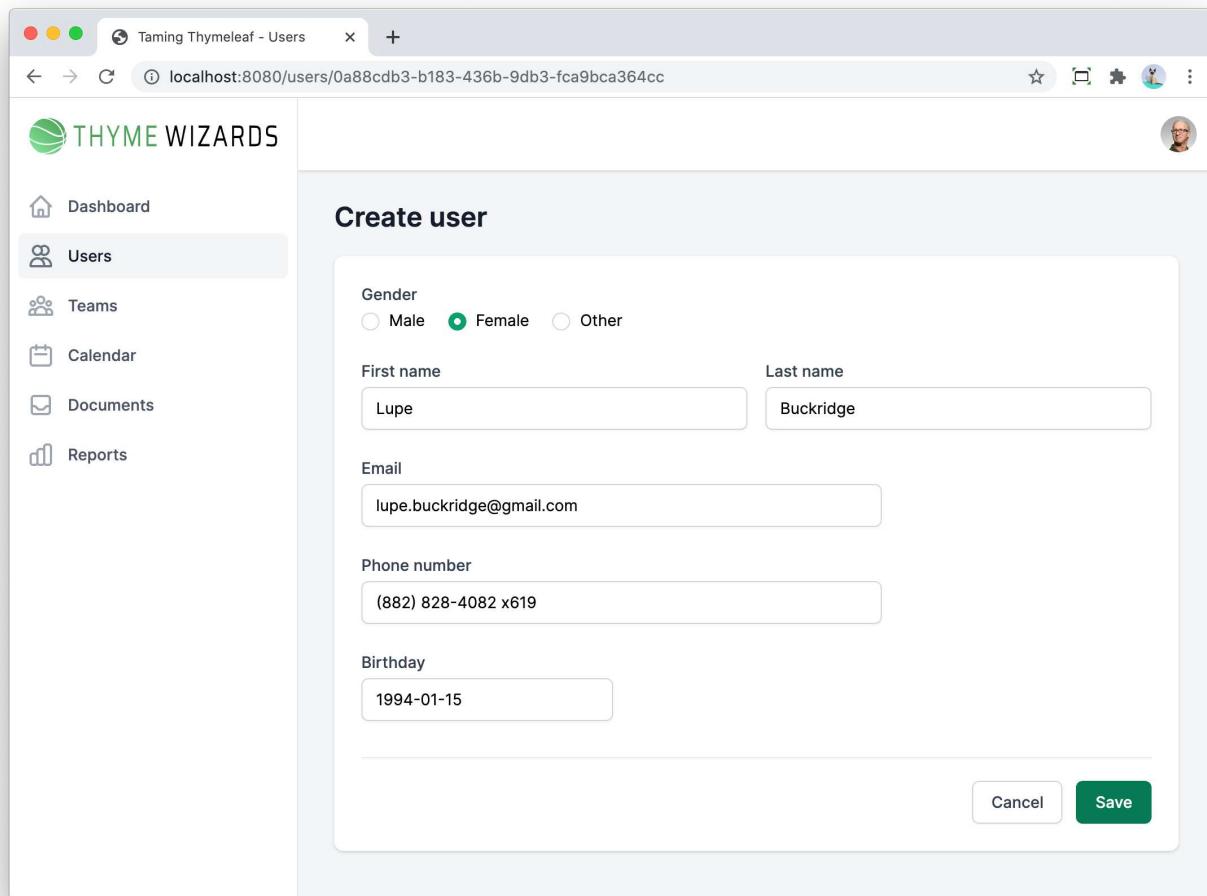


Figure 51. Link to a specific user shows the user details

This is already nice, but we are not done yet. Clicking 'Save' now would trigger the user creation, not the user editing. Stop the application and let's finish the implementation.

Update `edit.html` to take the `EditMode` into account:

1. Update `th:action` to either link to `/users/create` or `/users/{id}`:

```
<form id="user-form"
      th:object="${user}"
      th:action="${editMode?.name() == 'UPDATE'}?@{/users/{id}(id=${user.id})}:@{/users/create}"
      method="post">
```

2. Add the `version` field as a hidden input. We need to know the `version` value when the `POST` is done, so we can check if it still matches with the database:

```
...
<div th:replace="fragments/forms :: fielderrors"></div>
<div class="grid grid-cols-1 row-gap-6 col-gap-4 sm:grid-cols-6">
  <input type="hidden" th:field="*{version}">
```

```
th:if="${editMode?.name() == 'UPDATE'}">
```

```
...
```

3. Update the title to indicate if we are editing or creating:

```
<h1 class="text-2xl font-semibold text-gray-900"  
     th:text="${editMode?.name() ==  
'UPDATE'}?#{user.edit}:#{user.create}">Create user</h1>
```

4. Update the save button text to indicate if we are saving or creating:

```
<button type="submit"  
        class="ml-3 inline-flex justify-center py-2 px-4 border  
        border-transparent shadow-sm text-sm font-medium rounded-md text-  
        white bg-green-600 hover:bg-green-700 focus:outline-none focus:ring-2  
        focus:ring-offset-2 focus:ring-green-500"  
        th:text="${editMode?.name() == 'UPDATE'}?#{save}:#{create}">  
    Save  
</button>
```

The create form now has a 'Create' text on the save button:

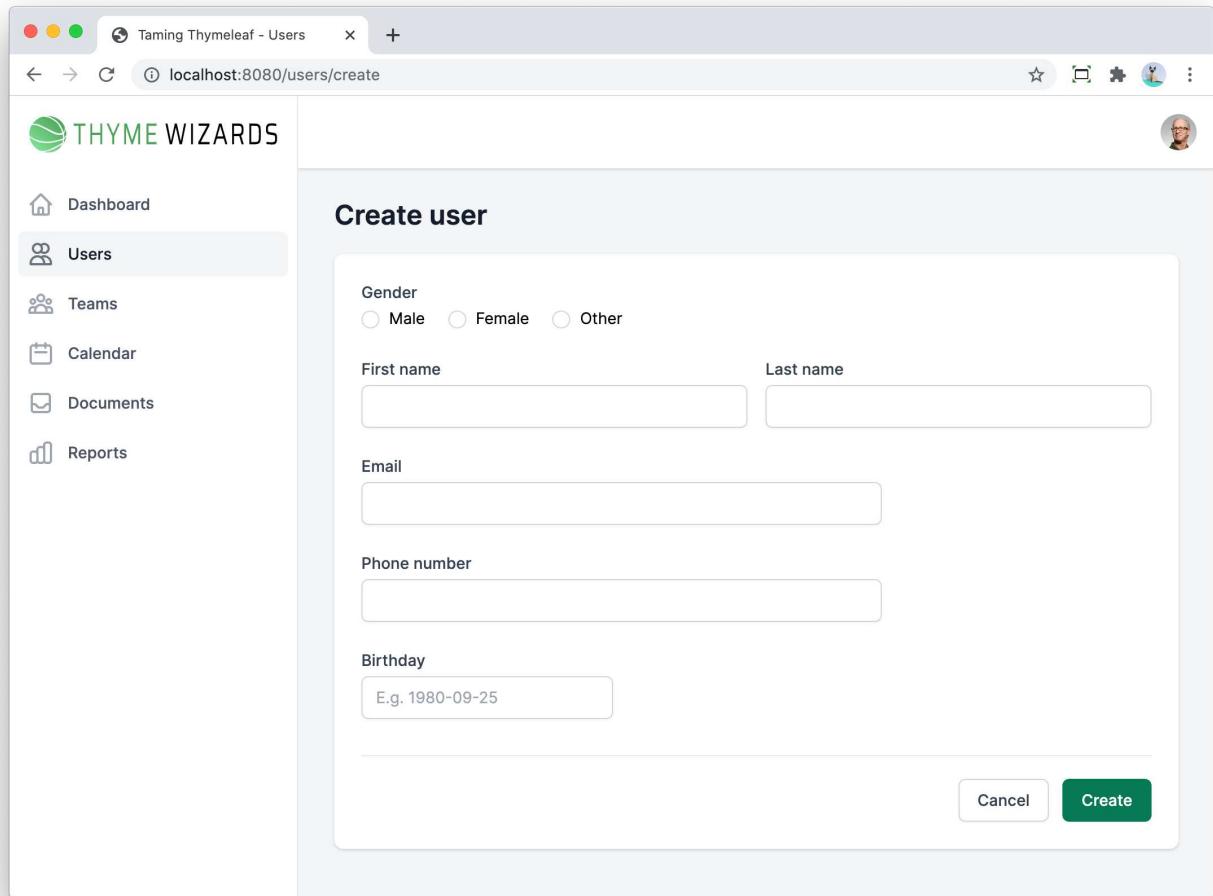


Figure 52. Create user showing 'Create' a text on the primary button

The edit form now has a proper title and 'Save' as primary action:

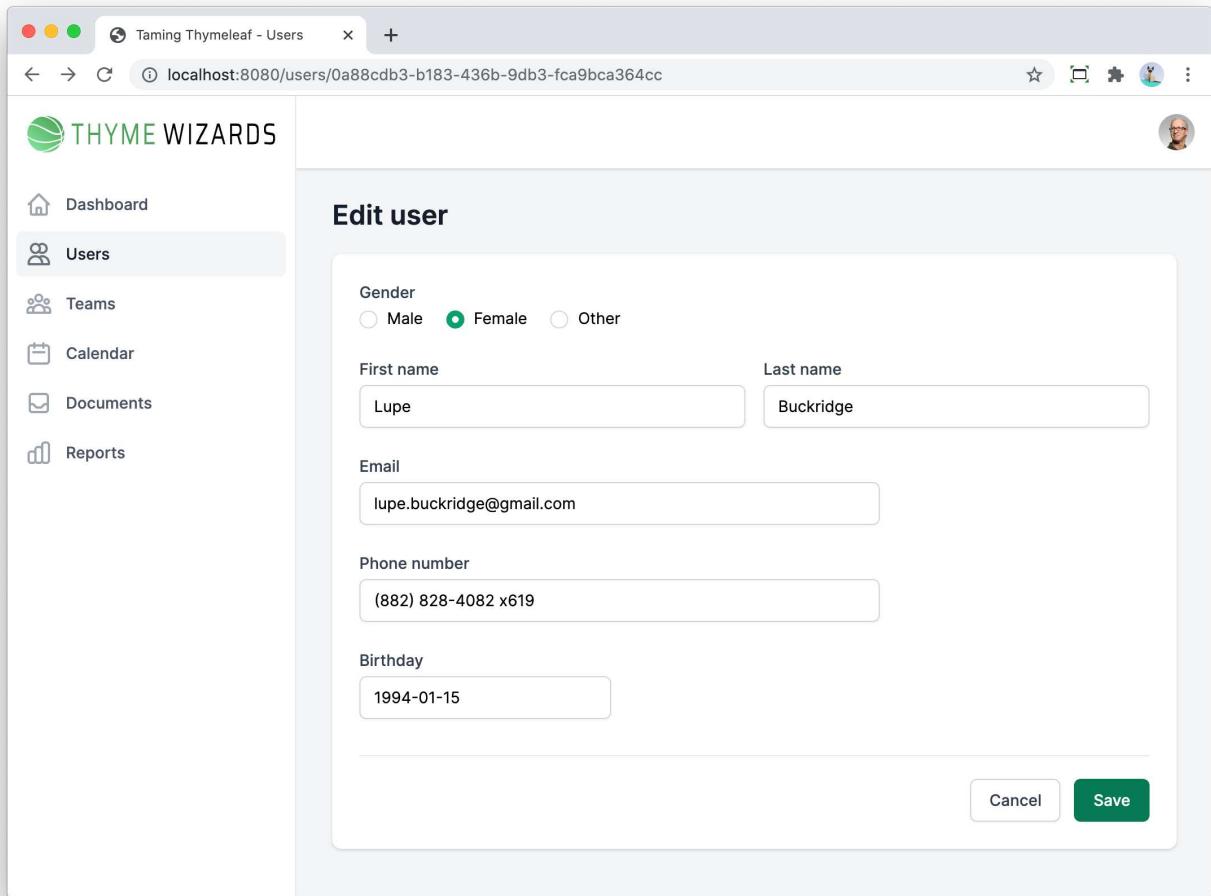


Figure 53. 'Edit user' title on form when editing a user

The next step is now implementing the actual save operation via a `POST` call in the controller:

`com.tamingthymeleaf.application.user.web.UserController`

```

@PostMapping("/{id}") ①
public String doEditUser(@PathVariable("id") UserId userId, ②
                        @Validated(ValidationGroupSequence.class)
@ModelAttribute("user") EditUserData formData, ③
                        BindingResult bindingResult, ④
                        Model model) {
    if (bindingResult.hasErrors()) { ⑤
        model.addAttribute("genders", List.of(Gender.MALE, Gender
.FEMALE, Gender.OTHER));
        model.addAttribute("EditMode", EditMode.UPDATE);
        return "users/edit";
    }

    service.editUser(userId, formData.toParameters()); ⑥

    return "redirect:/users"; ⑦
}

```

}

- ① Use the `@PostMapping` annotation to indicate that `POST` request to `/users/{id}` will call this controller method.
- ② Map the `{id}` part of the URL to the `UserId` variable via the `@PathVariable` annotation.
- ③ Inject the `EditUserData` instance that has the values from the HTML form.
- ④ Inject the `BindingResult` to check if there are validation errors.
- ⑤ If there are validation errors, show the HTML form again.
- ⑥ If there are no validation errors, call the `service` method to edit the user with the new values.
- ⑦ Redirect to the list of users after the update is done.

If we now try this out, we get this error when saving:

The screenshot shows a web application interface. On the left, a sidebar titled 'THYME WIZARDS' has 'Dashboard' selected. Under 'Users', the 'Edit user' page is active. The main content area has a heading 'Edit user'. A red error box contains the message: 'There were 1 errors with your submission' followed by a bullet point: 'There is already a user with the given email address.' Below this, there are input fields for 'First name' (Lupe), 'Last name' (Buckridge), 'Email' (lupe.buckridge@gmail.com), and 'Phone number' ((882) 828-4082 x619). The 'Email' field is highlighted with a red border and has an exclamation mark icon. Below the email field, a red message says 'There is already a user with the given email address.' There is also a 'Birthday' field containing '1994-01-15'.

Figure 54. User already exists validation error when trying to update a user

This is obviously not what we want. The reason we get this is because `EditUserData` extends from `CreateUserData` which is annotated with `@NotExistingUser(groups = ValidationGroupTwo.class)` and in the controller we ask for validation using this sequence:

```
@GroupSequence({Default.class, ValidationGroupOne.class,
ValidationGroupTwo.class})
```

```
public interface ValidationGroupSequence {
}
```

To fix this, we can rename `ValidationGroupSequence` to `CreateUserValidationGroupSequence` and create a 2nd sequence `EditUserValidationGroupSequence` that does not contain `ValidationGroupTwo`:

```
package com.tamingthymeleaf.application.user.web;

import
com.tamingthymeleaf.application.infrastructure.validation.ValidationGrou
pOne;

import javax.validation.GroupSequence;
import javax.validation.groups.Default;

@GroupSequence({Default.class, ValidationGroupOne.class})
public interface EditUserValidationGroupSequence {
}
```

We also moved them to the `com.tamingthymeleaf.application.user.web` package since they are now specific to the user stuff.

`com.tamingthymeleaf.application.user.web.UserController`

```
@PostMapping("/{id}")
public String doEditUser(@PathVariable("id") UserId userId,
                        @Validated(EditUserValidationGroupSequence
.class) @ModelAttribute("user") EditUserFormData formData, ①
                        BindingResult bindingResult,
                        Model model) {
    if (bindingResult.hasErrors()) {
        model.addAttribute("genders", List.of(Gender.MALE, Gender
.FEMALE, Gender.OTHER));
        model.addAttribute("EditMode", EditMode.UPDATE);
        return "users/edit";
    }

    service.editUser(userId, formData.toParameters());

    return "redirect:/users";
}
```

① Using `EditUserValidationGroupSequence` for validation of the edit of a user



This is probably a bit at the limit of what is still clear. Another way would be to copy the fields of `CreateUserData` into `EditUserData` and *not* have the latter extend from the former. This has the obvious drawback of code duplication, but will be easier to understand for the future reader of the code.

With these changes in place, the editing of a user works perfectly.

## 12.3. Refactoring to fragments

Our `edit.html` still has a lot of duplication going on per property. We should refactor this a bit and introduce a few fragments to reduce the duplication.

As a reminder, this is the HTML for a single input we currently have:

`src/main/resources/templates/users/edit.html`

```
<div class="sm:col-span-3">
    <label for="firstName" class="block text-sm font-medium text-gray-700">
        th:text="#{user.firstName}">
        First name
    </label>
    <div class="mt-1 relative rounded-md shadow-sm">
        <input id="firstName"
            type="text"
            th:field="*{firstName}"
            class="shadow-sm block w-full sm:text-sm border-gray-300 rounded-md"
            th:classappend="#{#fields.hasErrors('firstName')?'border-red-300 focus:border-red-300 focus:ring-red-500':'focus:ring-green-500 focus:border-green-500}'>
        <div th:if="#{#fields.hasErrors('firstName')}"
            class="absolute inset-y-0 right-0 pr-3 flex items-center pointer-events-none">
            <svg class="h-5 w-5 text-red-500" fill="currentColor"
            viewBox="0 0 20 20">
                <path fill-rule="evenodd"
                    d="M18 10a8 8 0 11-16 0 8 8 0 0116 0zm-7 4a1 1 0
11-2 0 1 1 0 012 0zm-1-9a1 1 0 00-1 1v4a1 1 0 102 0V6a1 1 0 00-1-1z"
                    clip-rule="evenodd"/>
            </svg>
        </div>
    </div>
    <p th:if="#{#fields.hasErrors('firstName')}"
        th:text="#{#strings.listJoin(#fields.errors('firstName'), ', ')}"
        class="mt-2 text-sm text-red-600" id="firstName-error">First name validation error message(s).</p>
```

```
</div>
```

Looking at the other inputs, we see these differences between all of them:

- Name of the field
- Name of the label
- Type of the field (plain text input vs email input)
- `class` used at the top-level (`sm:col-span-3`, `sm:col-span-4`, ...) to indicate how much room the input gets in the 6 column layout that makes up the form.
- Use of `placeholder` for the birthday.

We can add a new fragment to `templates/fragments/forms.html`:

```
<div th:fragment="textinput(labelText, fieldName, cssClass)">

    th:with="inputType=${inputType?:'text'},placeholder=${placeholder?:''}"
        th:class="${cssClass}">
        <label th:for ="${fieldName}" class="block text-sm font-medium text-gray-700">
            th:text ="${labelText}">
            Text input label
        </label>
        <div class="mt-1 relative rounded-md shadow-sm">
            <input th:id ="${fieldName}">
                th:type ="${inputType}">
                th:placeholder ="${placeholder}">
                th:field="*__${fieldName}__"
                class="shadow-sm block w-full sm:text-sm border-gray-300 rounded-md"

            th:classappend=" ${#fields.hasErrors(' __${fieldName}__')?'border-red-300
focus:border-red-300 focus:ring-red-500':'focus:ring-green-500
focus:border-green-500'}">
                <div th:if=" ${#fields.hasErrors(' __${fieldName}__')}>
                    class="absolute inset-y-0 right-0 pr-3 flex items-center pointer-events-none">
                        <svg class="h-5 w-5 text-red-500" fill="currentColor"
viewBox="0 0 20 20">
                            <path fill-rule="evenodd"
                                d="M18 10a8 8 0 11-16 0 8 8 0 0116 0zm-7 4a1 1 0
11-2 0 1 1 0 012 0zm-1-9a1 1 0 00-1 1v4a1 1 0 102 0V6a1 1 0 00-1-1z"
                                clip-rule="evenodd"/>
                        </svg>
                </div>
        </div>
    </div>
```

```

</div>
<p th:if="#{#fields.hasErrors('$_{#{fieldName}}_')}"
   th:text="#{#strings.listJoin(#fields.errors('$_{#{fieldName}}_'),
', ')}"
   class="mt-2 text-sm text-red-600" th:id="'$_{#{fieldName}}_+ '-
error'">Field validation error message(s).</p>
</div>

```

We created a `textinput` fragment with 3 required parameters:

- `labelText`: the (translated) name for the label to show to the user
- `fieldName`: the name of the field that this input should use. By using [Preprocessing](#), we can fill in the field name and then have Thymeleaf render the whole thing.
- `cssClass`: The CSS class to use on the top-level `<div>` for the layout of the input inside the form.

And 2 optional parameters:

- `inputType`: Allows to set the `type` of the `<input>`. If not set, defaults to `text`.
- `placeholder`: Allows to add a placeholder text. If not set, defaults to no placeholder.

We can now update `users/edit.html` to use the fragment:

```

<div th:replace="fragments/forms :: textinput(${user.firstName},
'firstName', 'sm:col-span-3')"></div>
<div th:replace="fragments/forms :: textinput(${user.lastName},
'lastName', 'sm:col-span-3')"></div>
<div th:replace="fragments/forms :: textinput(labelText=${user.email},
fieldName='email', cssClass='sm:col-span-4', inputType='email')"></div>
<div th:replace="fragments/forms :: textinput(${user.phoneNumber},
'phoneNumber', 'sm:col-span-4')"></div>
<div class="sm:col-span-2"></div>
<div th:replace="fragments/forms :: textinput(labelText=${user.birthday},
fieldName='birthday',
cssClass='sm:col-span-2',
placeholder=${user.birthday.placeholder})"></div>

```

Short and sweet! We went from around 120 lines of code to 6 and we made it a whole lot more readable in doing so.

We did not create a fragment for `gender` since it is the only use for a radio button input for now. We can always do this later using the same technique we used here.

## 12.4. Handling Optimistic Locking failure

We added a `version` field to our `User` entity to be able to use Optimistic Locking. We can quickly test if this works as follows:

- Open the application in 2 browsers (or tabs).
- Click the 'Edit' link for a user in 1 browser.
- Click the 'Edit' link for the same user in the other browser.
- Change something in the first browser and save.
- Without refreshing the 2nd browser, change something else there and save.

You will get something like this:

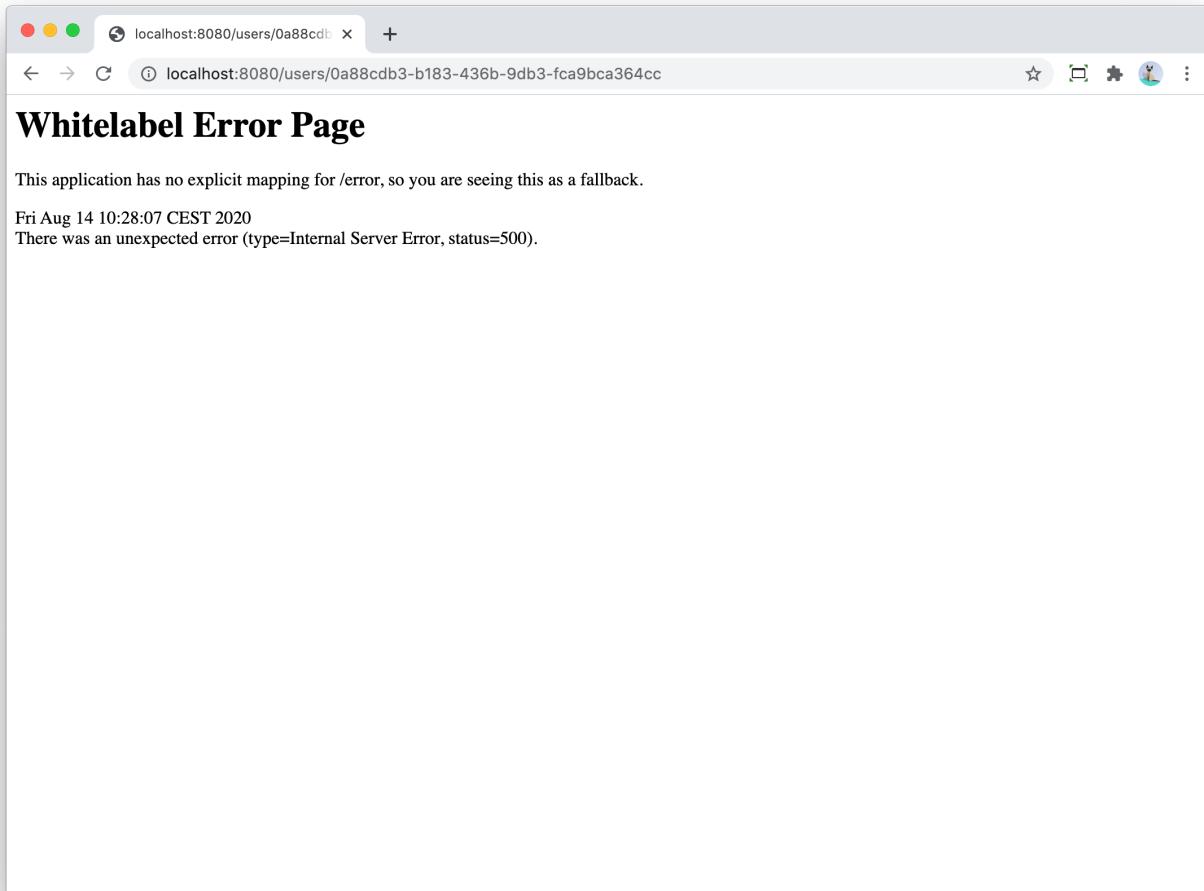


Figure 55. Spring Boot shows a default whitelabel error page for unhandled errors

Obviously not very user friendly. The log file will show that the optimistic locking kicked in:

```
org.springframework.orm.ObjectOptimisticLockingFailureException: Object
of class [com.tamingthymeleaf.application.user.User] with identifier
[0a88cdb3-b183-436b-9db3-fca9bca364cc]: optimistic locking failed
```

So, it is great that we did not get a lost update, but we should handle the error better.

To do this, we create an `@ControllerAdvice` annotated class that will handle the exception and show the appropriate view:

```
package com.tamingthymeleaf.application.infrastructure.web;
```

```

import org.springframework.dao.DataIntegrityViolationException;
import org.springframework.http.HttpStatus;
import org.springframework.orm.ObjectOptimisticLockingFailureException;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.servlet.ModelAndView;

import javax.servlet.http.HttpServletRequest;

@ControllerAdvice ①
public class GlobalControllerAdvice {

    @ResponseStatus(HttpStatus.CONFLICT) ②
    @ExceptionHandler({DataIntegrityViolationException.class,
ObjectOptimisticLockingFailureException.class}) ③
    public ModelAndView handleConflict(HttpServletRequest request,
Exception e) { ④
        ModelAndView result = new ModelAndView("error/409"); ⑤
        result.addObject("url", request.getRequestURL()); ⑥
        return result;
    }
}

```

- ① Annotate the class with `@ControllerAdvice` so that everything in this class will be applied to all controllers.
- ② Return status code 409 CONFLICT.
- ③ The `handleConflict` method should be called for any `DataIntegrityViolationException` or `ObjectOptimisticLockingFailureException`.
- ④ Inject the `HttpServletRequest` and the `Exception` into the method. See `ExceptionHandler Javadoc` for more information on all possible arguments that can be used.
- ⑤ Have Thymeleaf render the `error/409.html` page
- ⑥ Add the request URL as `url` in the model so the error page can use this.

With this in place, we can add our `error/409.html` template that show the error message to the user:

```

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
      layout:decorate="~{layout/layout}"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      lang="en">
<head>

```

```

<title th:text="#{error}">Error</title>
</head>
<body>
<!--/*@thymesVar id="url" type="String"-->
<div layout:fragment="page-content">
    <div class="max-w-7xl mx-auto px-4 sm:px-6 md:px-8">
        <div class="py-4">
            <div class="text-gray-500">
                <p th:text="#{error.version.conflict}" class="mb-6">
                    >Somebody else has edited the same thing as you.
                    Please reload the
                    page and redo your edit.</p>
                <a th:href="${url}" class="flex items-center text-sm
text-green-600 hover:text-green-900">
                    <svg viewBox="0 0 20 20" fill="currentColor"
class="w-4 h-4 mr-2">
                        <path fill-rule="evenodd"
                            d="M9.707 16.707a1 1 0 01-1.414 0l-6-6a1 1
0 01-1.414l6-6a1 1 0 011.414 1.414L5.414 9H17a1 1 0 0110 2H5.414l4.293
4.293a1 1 0 010 1.414z"
                        clip-rule="evenodd"></path>
                    </svg>
                    <span th:text="#{back.to.previous.page}">Back to
previous page</span> </a>
                </div>
            </div>
        </div>
    </div>
</body>
</html>

```

If we now repeat the same sequence to trigger the `ObjectOptimisticLockingFailureException`, we get:

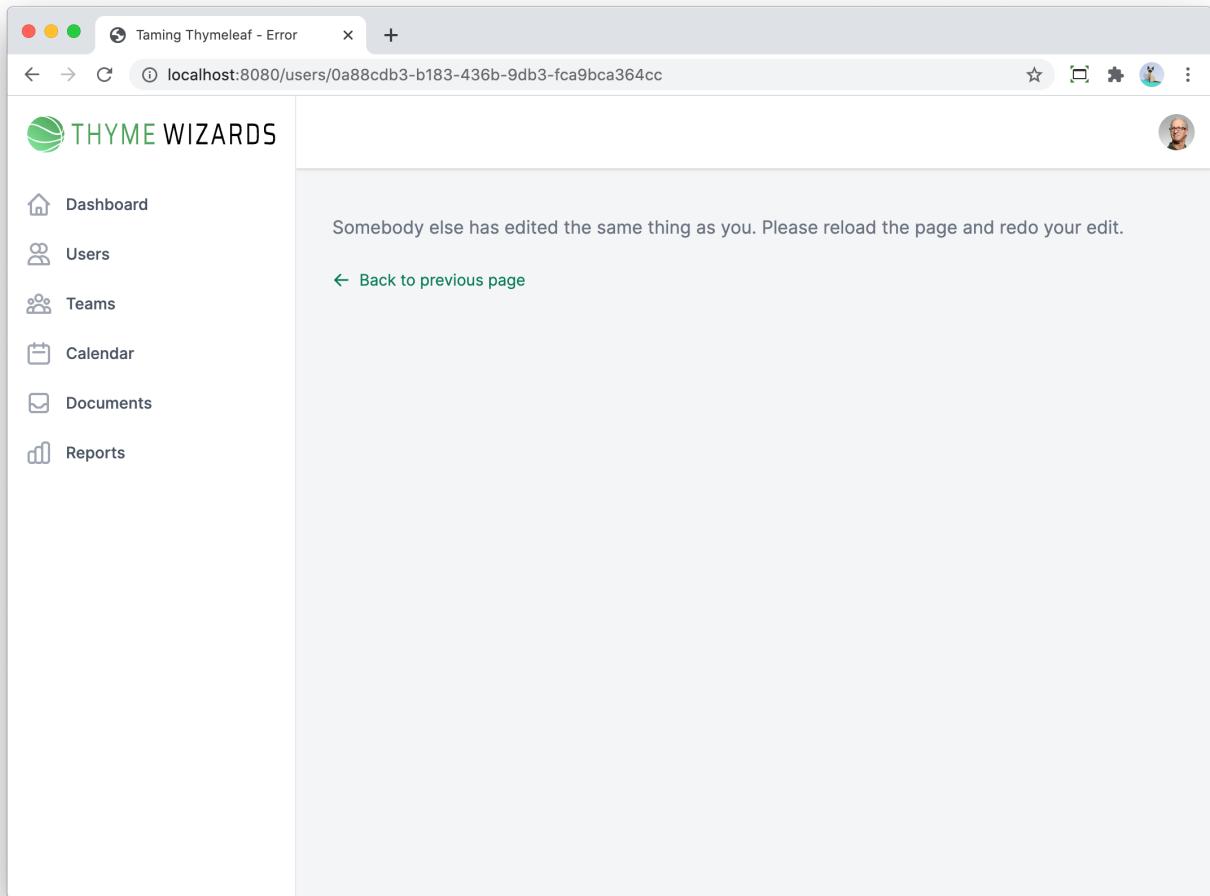


Figure 56. Custom error page for an optimistic locking failure

## 12.5. Custom error pages

Spring Boot has this whitelabel error page by default, which is quite ugly (intentionally probably so that people would add their own).

If we add a Thymeleaf template at `templates/error` with the error code as page name, then Spring Boot will automatically use that template when there is such an error code. For example, we can add a `404.html` page:

`src/main/resources/templates/error/404.html`

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
      layout:decorate="~{layout/layout}"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      lang="en">
<head>
    <title th:text="#{error}">Error</title>
</head>
<body>
<!--/*@thymesVar id="url" type="String"-->
```

```
<div layout:fragment="page-content">
    <div class="max-w-7xl mx-auto px-4 sm:px-6 md:px-8">
        <div class="py-4">
            <div class="text-gray-500">
                <p th:text="#{error.page.not.found}" class="mb-6">Page
not found.</p>
                <a th:href="@{}" class="flex items-center text-sm text-
green-600 hover:text-green-900">
                    <svg viewBox="0 0 20 20" fill="currentColor"
class="w-4 h-4 mr-2">
                        <path fill-rule="evenodd"
d="M9.707 16.707a1 1 0 01-1.414 0l-6-6a1 1
0 010-1.414l6-6a1 1 0 011.414 1.414L5.414 9H17a1 1 0 110 2H5.414l4.293
4.293a1 1 0 010 1.414z"
                        clip-rule="evenodd"></path>
                    </svg>
                    <span th:text="#{back.to.home.page}">Back to home
page</span> </a>
                </div>
            </div>
        </div>
    </div>
</body>
</html>
```

If you access a URL that does not exist like <http://localhost:8080/nonexisting>, there is a nice error message:

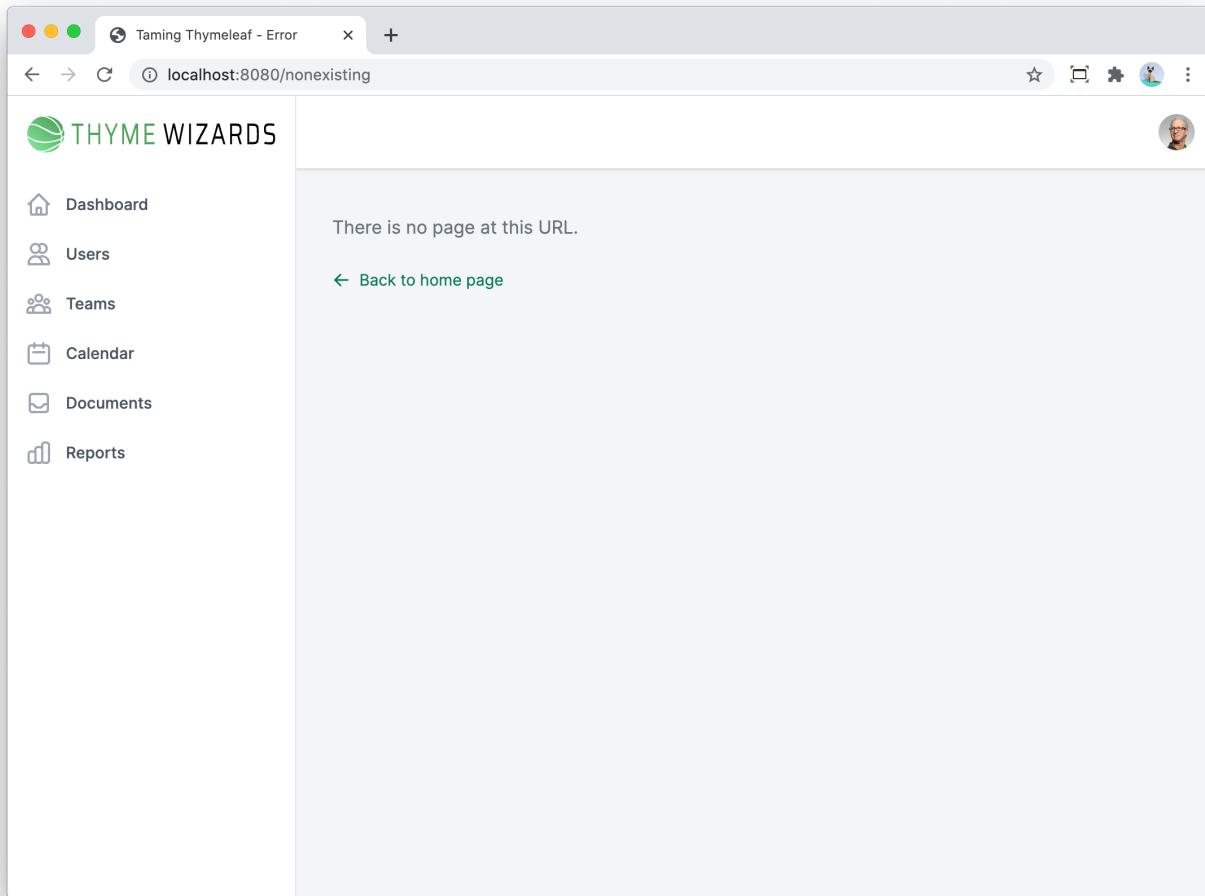


Figure 57. Custom error page for a 404 NOT FOUND error

If you want to have a single error page for multiple error codes, you can do that as well. For example, to have a single template to use for all errors in the **500-599** range, you can use **templates/error/5xx.html** as template name.

As an example, consider this template:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
      layout:decorate="~{layout/layout}"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      lang="en">
<head>
    <title th:text="#{error}">Error</title>
</head>
<body>
<!--/*@thymesVar id="url" type="String"-->
<div layout:fragment="page-content">
    <div class="max-w-7xl mx-auto px-4 sm:px-6 md:px-8">
        <div class="py-4">
            <div class="text-gray-500">
```

```

        <div class="mb-4">There was an error, please try again.
Contact the administrator if the problem
persists.
</div>
<a th:href="@{}" class="flex items-center text-sm text-
green-600 hover:text-green-900">
    <svg viewBox="0 0 20 20" fill="currentColor"
class="w-4 h-4 mr-2">
        <path fill-rule="evenodd"
d="M9.707 16.707a1 1 0 01-1.414 0l-6-6a1 1
0 01-1.414l6-6a1 1 0 011.414 1.414L5.414 9H17a1 1 0 110 2H5.414l4.293
4.293a1 1 0 010 1.414z"
        clip-rule="evenodd"></path>
    </svg>
    <span th:text="#{back.to.home.page}">Back to home
page</span> </a>
    <th:block th:if="${exception}">
        <div class="mb-2 mt-6 text-xl">Technical
Information</div>
        <div class="flex mb-2">
            <div class="mr-2">Error Code:</div>
            <div th:text="${status}" class="font-
mono"></div>
        </div>
        <div class="flex mb-2">
            <div class="mr-2">Exception:</div>
            <div th:text="${exception}" class="font-
mono"></div>
        </div>
        <div th:if="${trace}" class="mb-1">Stack
trace:</div>
        <div th:text="${trace}" class="ml-2 font-mono text-
sm"></div>
    </th:block>
</div>
</div>
</body>
</html>

```

To test this behaviour, you can add this method to the `UserController` for example:

```
@GetMapping("/ex")
```

```
public String throwException() {
    throw new RuntimeException("This is a fake exception for
testing");
}
```

The browser will now render <http://localhost:8080/users/ex> as follows:

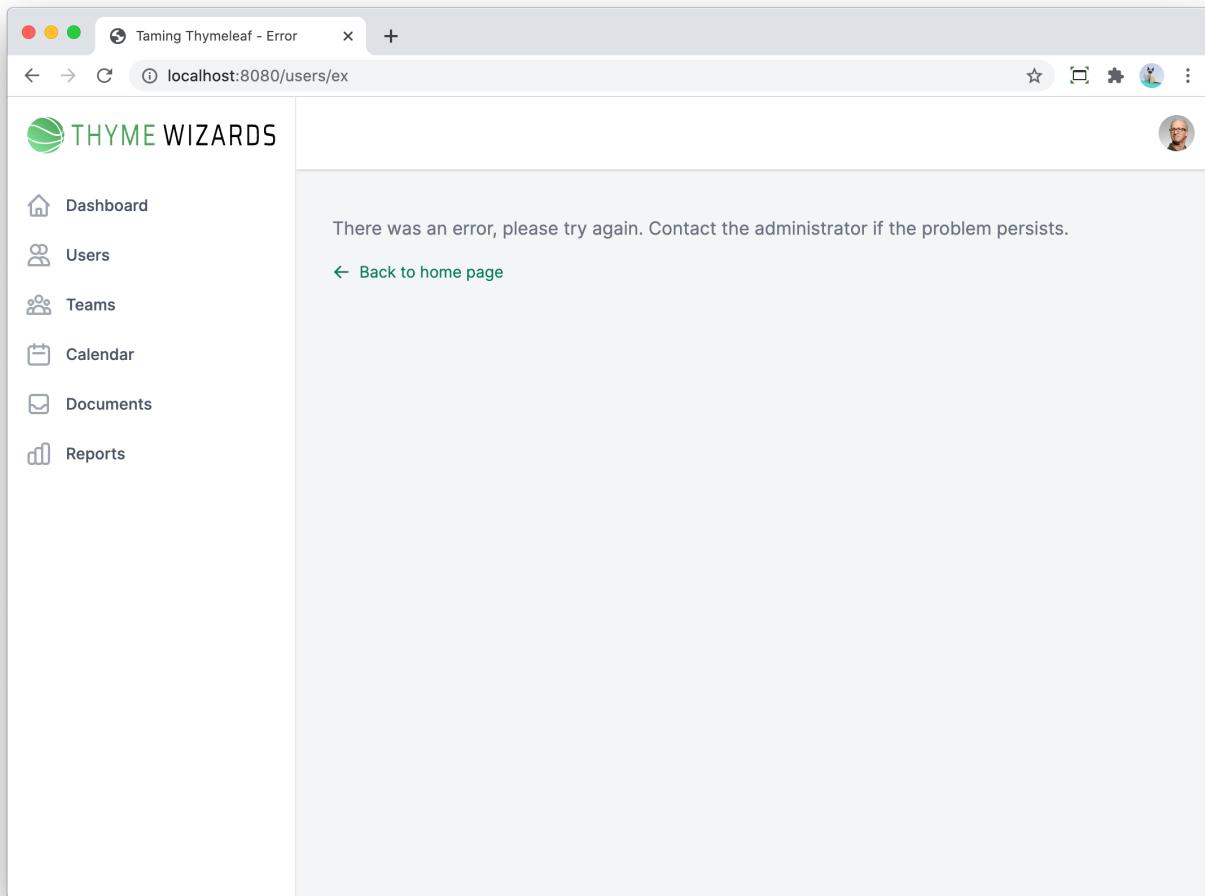


Figure 58. Custom wildcard error page

It does not show the 'Technical Information' block. This is because Spring Boot will *not* include the `exception` or `trace` variables by default for the template to render. We can change this behaviour with the following properties:

`src/main/resources/application.properties`

```
server.error.include-exception=true
server.error.include-stacktrace=always
```

Now, the template will render as:

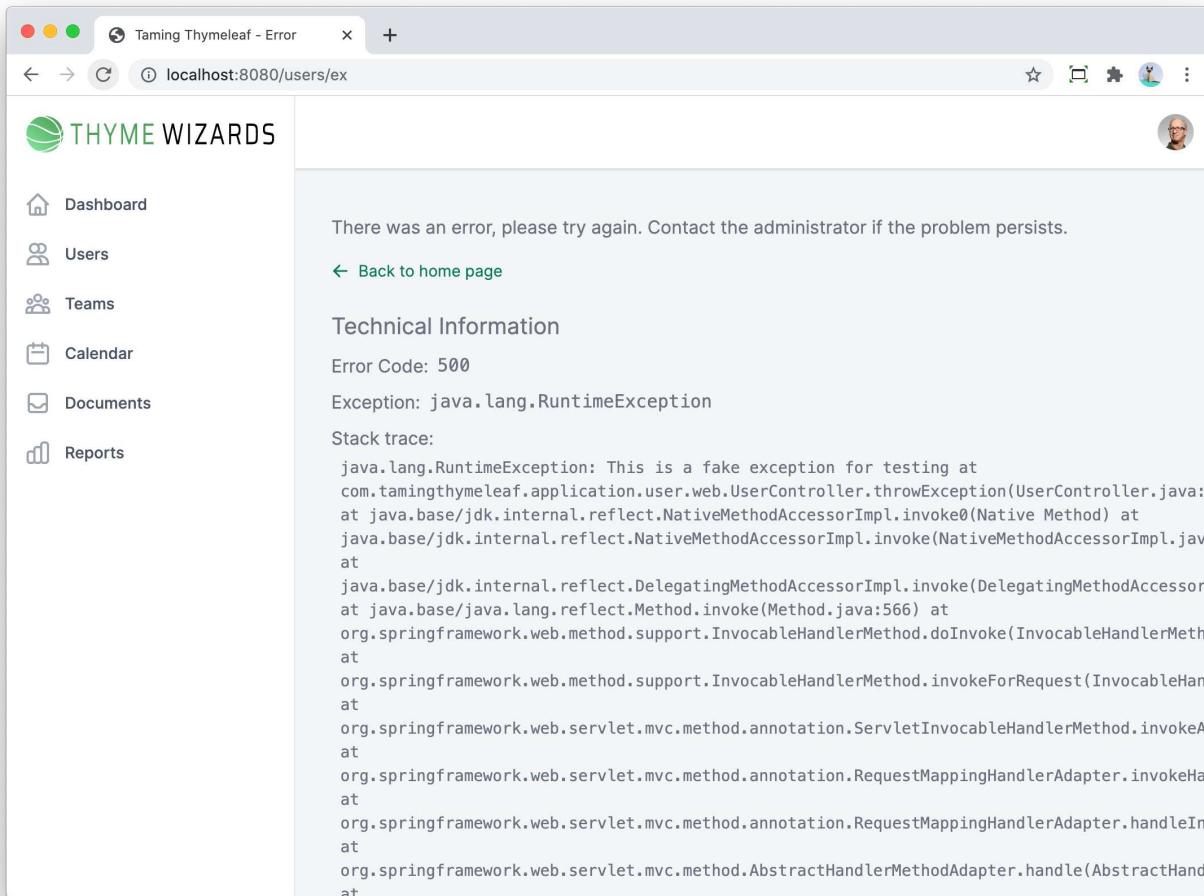


Figure 59. Custom error page showing exception and stacktrace information



In most cases, you will set those properties only in a development or staging environment. Don't show these technical details in production. This can be easily done by only setting those properties in an `application-dev.properties` and `application-staging.properties` file, but not in the `application-prod.properties`.

## 12.6. Summary

In this chapter, you learned:

- How to allow a user to edit properties of an entity.
- How to properly handle optimistic locking.
- How to add custom error pages for various error status codes.

# Chapter 13. Implement deletion of an entity

After implementing create, read and update, we have come to the last part of the CRUD formation: Delete

Web browsers only support **GET** and **POST**. We don't get to use all the other **HTTP request methods** (or HTTP verbs as they are sometimes called) that REST API developers can use.

Fortunately, we have 2 options to work around this limitation:

- Use a dedicated URL. For example, we can allow a **POST** on `/users/<id>/delete` to delete a user.
- Do a **POST** with the “real” method as an additional parameter (modeled as a hidden input field in an HTML form).



There is also the option to use JavaScript to call **DELETE** on a REST API endpoint (e.g. `/api/users/<id>`), but that would lead us too far.

## 13.1. Using a dedicated URL

This first example will use a dedicated URL as it requires no special support from Spring Boot to make it work.

We can just a new **PostMapping** to **UserController**:

```
com.tamingthymeleaf.application.user.web.UserController
```

```
@PostMapping("/{id}/delete")
public String doDeleteUser(@PathVariable("id") UserId userId) {
    service.deleteUser(userId);

    return "redirect:/users";
}
```

The controller just redirects to the **UserService**:

```
com.tamingthymeleaf.application.user.UserServiceImpl
```

```
@Override
public void deleteUser(UserId userId) {
    repository.deleteById(userId); ①
}
```

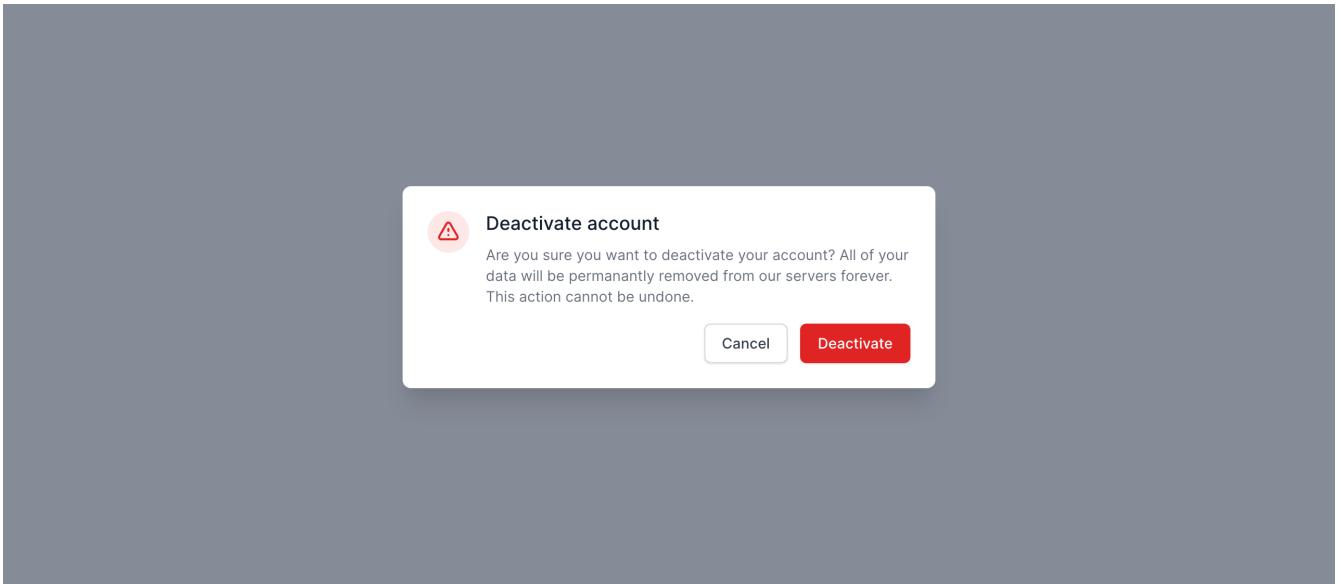
① Use `CrudRepository.deleteById()` to delete the user

Now, on the HTML side, we do have a lot of work. The goal is to add 'Delete' links on each row next to the 'Edit' link that is already there.

When it is pressed, a modal should ask for confirmation and once confirmed, the user should get

deleted.

For the design, we will use 'Simple alert' from the modals section in Tailwind UI:



The modal has a semi-transparent backdrop that will need to cover the complete application. To make that possible, we need to add a new layout-hook in our `templates/layout/layout.html` file. If we would use the existing `page-content` layout-hook, we would not cover the complete application.

`src/main/resources/templates/layout/layout.html`

```
...
<div layout:fragment="modals-content">
</div>
...
```

Now we can edit `templates/users/list.html` to use that `modals-content` layout fragment and copy the modal code:

```
<div layout:fragment="modals-content">
    <div class="fixed z-10 inset-0 overflow-y-auto">
        <div class="flex items-end justify-center min-h-screen pt-4 px-4 pb-20 text-center sm:block sm:p-0">
            <!--
                Background overlay, show/hide based on modal state.

                Entering: "ease-out duration-300"
                From: "opacity-0"
                To: "opacity-100"
                Leaving: "ease-in duration-200"
                From: "opacity-100"
                To: "opacity-0"
            -->
```

```

        <div class="fixed inset-0 transition-opacity" aria-
hidden="true">
            <div class="absolute inset-0 bg-gray-500 opacity-
75"></div>
        </div>

        <!-- This element is to trick the browser into centering the
modal contents. -->
        <span class="hidden sm:inline-block sm:align-middle sm:h-
screen" aria-hidden="true">⋮</span>
        <!--
            Modal panel, show/hide based on modal state.

            Entering: "ease-out duration-300"
            From: "opacity-0 translate-y-4 sm:translate-y-0
sm:scale-95"
                To: "opacity-100 translate-y-0 sm:scale-100"
            Leaving: "ease-in duration-200"
            From: "opacity-100 translate-y-0 sm:scale-100"
            To: "opacity-0 translate-y-4 sm:translate-y-0 sm:scale-
95"
        -->
        <div class="inline-block align-bottom bg-white rounded-lg
px-4 pt-5 pb-4 text-left overflow-hidden shadow-xl transform transition-
all sm:my-8 sm:align-middle sm:max-w-lg sm:w-full sm:p-6" role="dialog"
aria-modal="true" aria-labelledby="modal-headline">
            <div class="sm:flex sm:items-start">
                <div class="mx-auto flex-shrink-0 flex items-center
justify-center h-12 w-12 rounded-full bg-red-100 sm:mx-0 sm:h-10 sm:w-
10">
                    <!-- Heroicon name: outline/exclamation -->
                    <svg class="h-6 w-6 text-red-600"
xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0 24 24"
stroke="currentColor" aria-hidden="true">
                        <path stroke-linecap="round" stroke-
linejoin="round" stroke-width="2" d="M12 9v2m0 4h.01m-6.938
4h13.856c1.54 0 2.502-1.667 1.732-3L13.732 4c-.77-1.333-2.694-1.333-
3.464 0L3.34 16c-.77 1.333.192 3 1.732 3z" />
                    </svg>
                </div>
                <div class="mt-3 text-center sm:mt-0 sm:ml-4
sm:text-left">
                    <h3 class="text-lg leading-6 font-medium text-
gray-900" id="modal-headline">
                        Deactivate account

```

```
</h3>
<div class="mt-2">
    <p class="text-sm text-gray-500">
        Are you sure you want to deactivate your
        account? All of your data will be permanently removed from our servers
        forever. This action cannot be undone.
    </p>
    </div>
</div>
<div class="mt-5 sm:mt-4 sm:flex sm:flex-row-reverse">
    <button type="button" class="w-full inline-flex
justify-center rounded-md border border-transparent shadow-sm px-4 py-2
bg-red-600 text-base font-medium text-white hover:bg-red-700
focus:outline-none focus:ring-2 focus:ring-offset-2 focus:ring-red-500
sm:ml-3 sm:w-auto sm:text-sm">
        Deactivate
    </button>
    <button type="button" class="mt-3 w-full inline-flex
justify-center rounded-md border border-gray-300 shadow-sm px-4 py-2 bg-
white text-base font-medium text-gray-700 hover:bg-gray-50
focus:outline-none focus:ring-2 focus:ring-offset-2 focus:ring-indigo-
500 sm:mt-0 sm:w-auto sm:text-sm">
        Cancel
    </button>
</div>
</div>
</div>
</div>
```

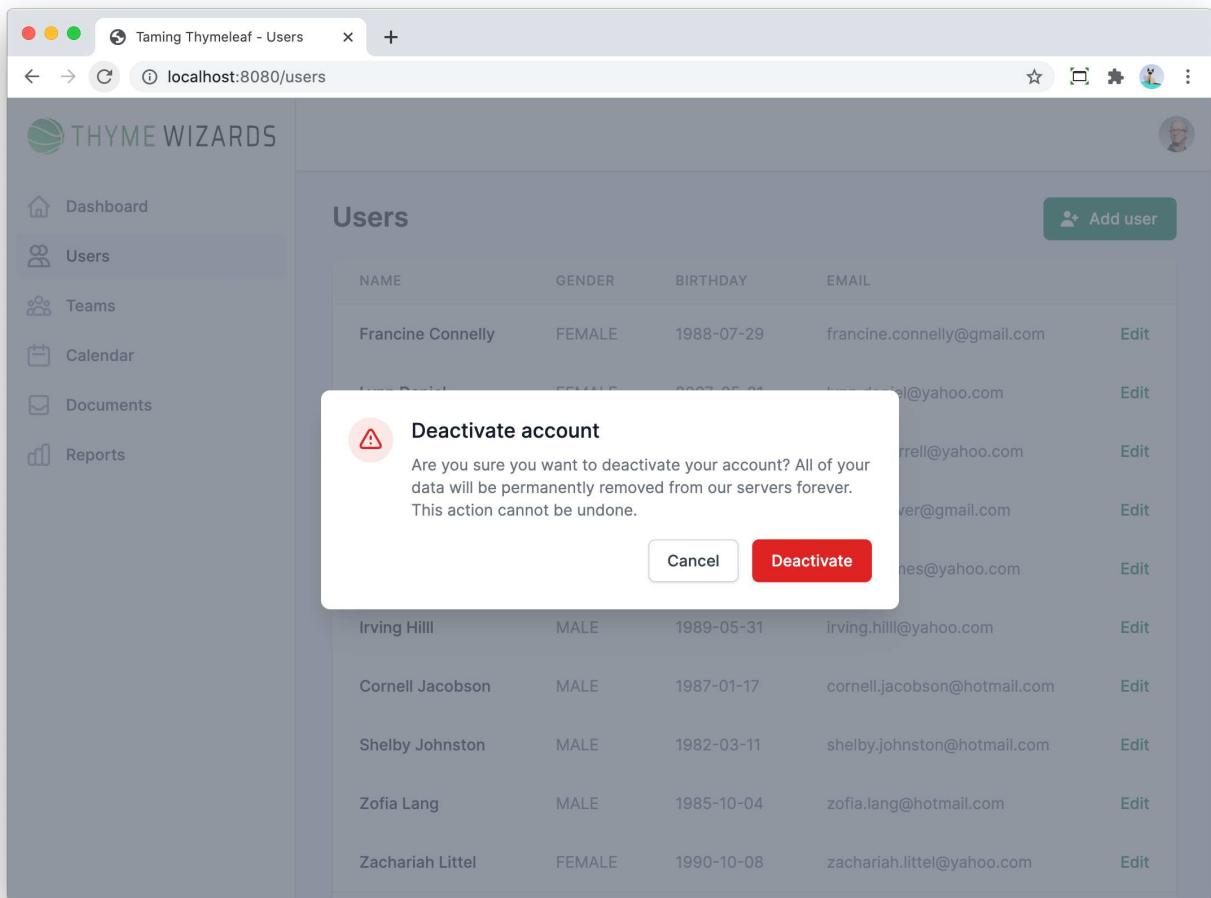


Figure 60. Modal on top of current application

This already looks good, but it does not do anything useful. Let's change that.

We start by adding a 'Delete' link in the table.

Add a header cell:

```
src/main/resources/templates/users/list.html
```

```
<th th:replace="fragments/table :: header('')"></th>
```

Add a data cell:

```
src/main/resources/templates/users/list.html
```

```
<td th:x-data="|{name: '${user.userName.fullName}', deleteUrl: '/users/${user.id.asString()}/delete'}|"
    class="px-6 py-4 whitespace-no-wrap text-right text-sm leading-5 font-medium">
    <a href="javascript:void(0)" class="text-green-600 hover:text-green-900" @click="$dispatch('open-delete-modal', {name, deleteUrl})">
        Delete</a>

```

```
</td>
```

There is a lot going on, so let's break it down:

- `th:x-data` will have Thymeleaf render an `x-data` attribute containing the data that will be sent by AlpineJS from the 'Delete' link to the modal dialog.
- The `href` element on the `<a>` tag uses `javascript:void(0)` to avoid that browser goes to another page. We just want it to trigger the `click` event.
- `@click` is the AlpineJS way to registering a callback when the `click` event is triggered. When it is, we dispatch an `open-delete-modal` event and give it the `name` and `deleteUrl` from the `x-data` attribute. The `name` is used to show the name of the user that will be deleted. The `deleteUrl` is the URL that we will `POST` too, to trigger the actual delete of the user.

Next, we need to make the modal itself dynamic:

```
<div layout:fragment="modals-content" x-data=
"modalDeleteConfirmation()"> ①
    <div x-show="isVisible()"
        x-cloak
        @open-delete-modal.window="showModal($event)"
        class="fixed z-10 inset-0 overflow-y-auto"> ②
        <div class="flex items-end justify-center min-h-screen pt-4 px-4
pb-20 text-center sm:block sm:p-0">
            <!--
                Background overlay
            -->
            <div class="fixed inset-0 transition-opacity"
                aria-hidden="true"
                x-show="isVisible()"
                x-transition:enter="ease-out duration-300"
                x-transition:enter-start="opacity-0"
                x-transition:enter-end="opacity-100"
                x-transition:leave="ease-in duration-200"
                x-transition:leave-start="opacity-100"
                x-transition:leave-end="opacity-0">
                <div class="absolute inset-0 bg-gray-500 opacity-
75"></div>
            </div>

            <!-- This element is to trick the browser into centering the
            modal contents. -->
            <span class="hidden sm:inline-block sm:align-middle sm:h-
screen" aria-hidden="true"></span>#8203;
            <!--
                Modal panel
            -->
        </div>
    </div>
```

```

-->
<div class="inline-block align-bottom bg-white rounded-lg
px-4 pt-5 pb-4 text-left overflow-hidden shadow-xl transform transition-
all sm:my-8 sm:align-middle sm:max-w-lg sm:w-full sm:p-6"
      role="dialog" aria-modal="true" aria-labelledby="modal-
headline"
      x-show="isVisible()"
      x-transition:enter="ease-out duration-300"
      x-transition:enter-start="opacity-0 translate-y-4
sm:translate-y-0 sm:scale-95"
      x-transition:enter-end="opacity-100 translate-y-0
sm:scale-100"
      x-transition:leave="ease-in duration-200"
      x-transition:leave-start="opacity-100 translate-y-0
sm:scale-100"
      x-transition:leave-end="opacity-0 translate-y-4
sm:translate-y-0 sm:scale-95"> ③
  <div class="sm:flex sm:items-start">
    <div class="mx-auto flex-shrink-0 flex items-center
justify-center h-12 w-12 rounded-full bg-red-100 sm:mx-0 sm:h-10 sm:w-
10">
      <!-- Heroicon name: outline/exclamation -->
      <svg class="h-6 w-6 text-red-600" fill="none"
viewBox="0 0 24 24" stroke="currentColor">
        <path stroke-linecap="round" stroke-
linejoin="round" stroke-width="2"
          d="M12 9v2m0 4h.01m-6.938
4h13.856c1.54 0 2.502-1.667 1.732-3L13.732 4c-.77-1.333-2.694-1.333-
3.464 0L3.34 16c-.77 1.333.192 3 1.732 3z"/>
      </svg>
    </div>
    <div class="mt-3 text-center sm:mt-0 sm:ml-4
sm:text-left">
      <h3 class="text-lg leading-6 font-medium text-
gray-900" id="modal-headline">
        Delete user
      </h3>
      <div class="mt-2">
        <p class="text-sm text-gray-500"> ④
          Are you sure you want to delete user
<span class="italic"
x-text="getName()"></span>?
        </p>
      </div>
    </div>
  </div>
</div>

```

```

        </div>
    </div>
    <form id="deleteForm" enctype="multipart/form-data"
method="post"
            x-bind:action="getDeleteUrl()"> ⑤
            <div class="mt-5 sm:mt-4 sm:flex sm:flex-row-
reverse">
                <span class="flex w-full rounded-md shadow-sm sm:ml-3 sm:w-
auto">
                    <button type="submit"
                            class="w-full inline-flex justify-center rounded-md
border border-transparent shadow-sm px-4 py-2 bg-red-600 text-base font-
medium text-white hover:bg-red-700 focus:outline-none focus:ring-2
focus:ring-offset-2 focus:ring-red-500 sm:ml-3 sm:w-auto sm:text-sm">
                        Delete
                    </button>
                </span>
                <span class="mt-3 flex w-full rounded-md shadow-
sm sm:mt-0 sm:w-auto">
                    <button type="button"
                            @click="hideModal"
                            class="mt-3 w-full inline-flex justify-center rounded-
md border border-gray-300 shadow-sm px-4 py-2 bg-white text-base font-
medium text-gray-700 hover:bg-gray-50 focus:outline-none focus:ring-2
focus:ring-offset-2 focus:ring-indigo-500 sm:mt-0 sm:w-auto sm:text-sm">
                        ⑥
                        Cancel
                    </button>
                </span>
            </div>
            </form>
        </div>
    </div>
</div>

```

① We make the `<div>` an AlpineJS component By using `x-data="modalDeleteConfirmation()"`. See the JavaScript of that function below.

② 3 important things here:

- `x-show="isVisible()"` binds the visibility of the element to the result of the `isVisible()` method.
- `x-cloak` avoids a flash of the modal while the JavaScript is being loaded.
- `@open-delete-modal.window="showModal($event)"`: this installs a global listener for the `open-delete-modal` event that we dispatch from the 'Delete' links in the table. By passing the `$event`, we will be able to access the details of which delete link was pressed so we know what

user to delete.

- ③ We again have `x-show="isVisible()"` here. The reason we don't have it just on the top `<div>` is to make the `x-transition` attributes work. They ensure we have some nice fade-in and fade-out effects when showing or hiding the modal dialog.
- ④ `x-text="getName()"` will replace the inner HTML of the `<span>` with the text returned by the method. This is very similar to `th:text`. The big difference is that `th:text` is processed by Thymeleaf, so it is done once when the page is rendered. We now need the text to change dynamically with JavaScript each time a different 'Delete' link is clicked, so we need to use AlpineJS' `x-text` instead.
- ⑤ We need a `<form>` to do the `POST` request. Since we need to update the URL for each to-be-deleted user, we also need a JavaScript based way of updating the target URL. This is done via the `x-bind:action="getDeleteUrl()"` attribute. The AlpineJS instruction `x-bind` can target any attribute. Here we use it to update the `action` of the form to have the correct URL to delete the user.
- ⑥ For the 'Cancel' button, we hide the modal when the `click` event happens.

Because we have not added `Security` yet, we do the POST without CSRF protection. Once we add security in the next chapter, we will need to add this to our form to make it work:



```
<input type="hidden"
       th:name="${_csrf.parameterName}"
       th:value="${_csrf.token}"/> ①
```

- ① Thymeleaf normally adds a hidden input with a CSRF token to all forms automatically, but *only* if you set a `th:action` on the form. Because we don't use `th:action`, but set the action in JavaScript, we need to manually add a hidden input with the CSRF token. We can use the `_csrf` value that is always present to do so.

As became apparent during the explanation of the above HTML, we need some JavaScript to make it all work:

```
<th:block layout:fragment="page-scripts">
<script>
    function modalDeleteConfirmation() {
        return {
            show: false,
            name: '',
            deleteUrl: '',
            hideModal() {
                this.show = false;
            },
            isVisible() {
                return this.show === true;
            },
            getName() {
```

```

        return this.name;
    },
    getDeleteUrl() {
        return this.deleteUrl;
    },
    showModal($event) { ①
        this.name = $event.detail.name;
        this.deleteUrl = $event.detail.deleteUrl;
        this.show = true;
    }
};

</script>
</th:block>

```

- ① Using the passed in `$event` parameter, we can extract the `name` and `deleteUrl` properties. Due to the use of `x-show`, `x-text` and `x-bind` in the HTML, the modal dialog will show the name of the to-be-deleted user with the correct `<form>` target URL in place as soon as the `showModal()` method is called.

With all this, we now have a list of users with a 'Delete' link:

The screenshot shows a web application interface titled "Taming Thymeleaf - Users" at the URL "localhost:8080/users". The left sidebar has a "THYME WIZARDS" logo and navigation links: Dashboard, Users (which is selected), Teams, Calendar, Documents, and Reports. The main content area is titled "Users" and contains a table with the following data:

NAME	GENDER	BIRTHDAY	EMAIL	Edit	Delete
Francine Connelly	FEMALE	1988-07-29	francine.connelly@gmail.com	<a href="#">Edit</a>	<a href="#">Delete</a>
Lynn Daniel	FEMALE	2007-05-21	lynn.daniel@yahoo.com	<a href="#">Edit</a>	<a href="#">Delete</a>
Gordon Farrell	FEMALE	1981-11-13	gordon.farrell@yahoo.com	<a href="#">Edit</a>	<a href="#">Delete</a>
Efrain Glover	FEMALE	1994-08-14	efrain.glover@gmail.com	<a href="#">Edit</a>	<a href="#">Delete</a>
Kelley Grimes	MALE	2008-04-24	kelley.grimes@yahoo.com	<a href="#">Edit</a>	<a href="#">Delete</a>
Irving Hill	MALE	1989-05-31	irving.hill@yahoo.com	<a href="#">Edit</a>	<a href="#">Delete</a>
Cornell Jacobson	MALE	1987-01-17	cornell.jacobson@hotmail.com	<a href="#">Edit</a>	<a href="#">Delete</a>
Shelby Johnston	MALE	1982-03-11	shelby.johnston@hotmail.com	<a href="#">Edit</a>	<a href="#">Delete</a>
Zofia Lang	MALE	1985-10-04	zofia.lang@hotmail.com	<a href="#">Edit</a>	<a href="#">Delete</a>
Zachariah Littel	FEMALE	1990-10-08	zachariah.littel@yahoo.com	<a href="#">Edit</a>	<a href="#">Delete</a>

Figure 61. List of users with delete link

Clicking on the 'Delete' link shows the confirmation dialog:

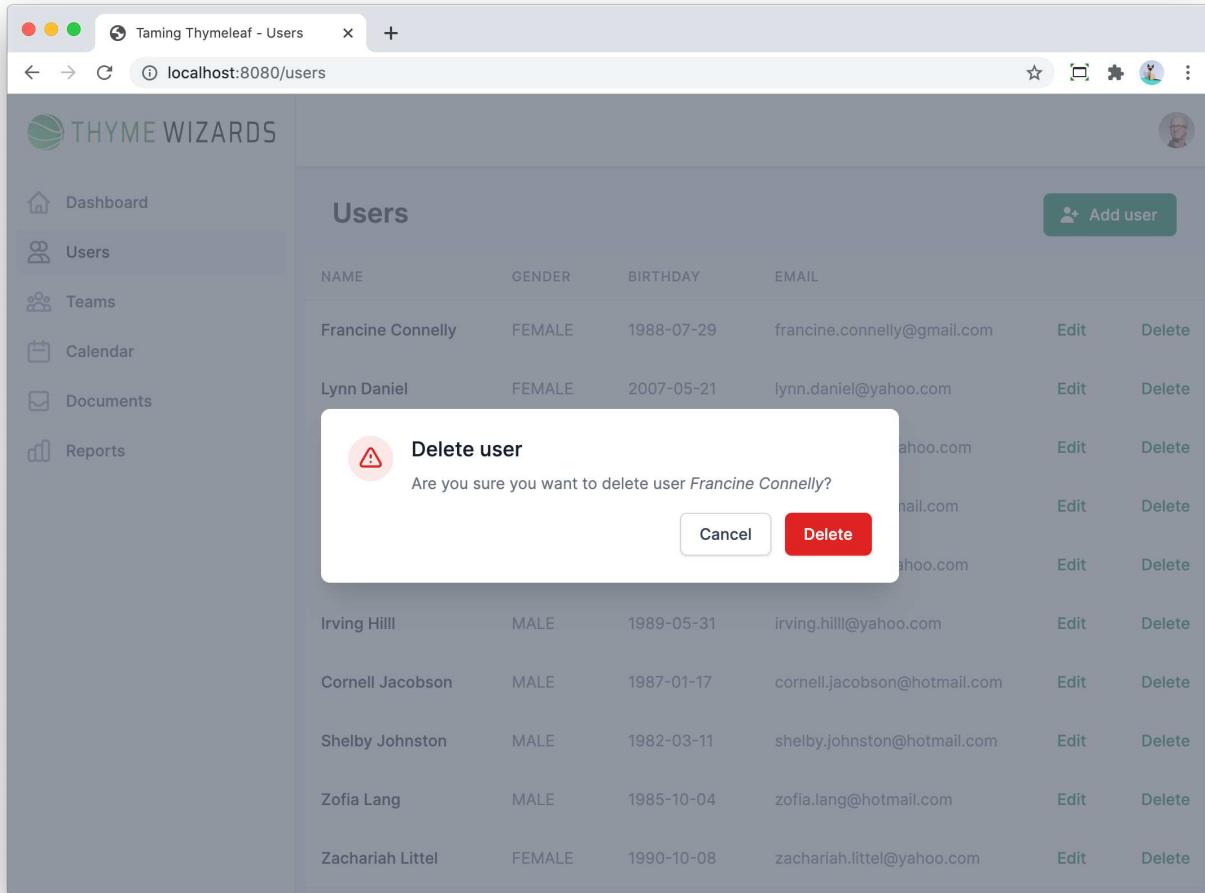


Figure 62. Modal confirmation dialog to delete a user

Confirming the question will delete the user in the database and redirect the browser to the list of users.

## 13.2. Using the DELETE HTTP method

The previous section used `/users/<id>/delete` as the URL to `POST` to. We will show an alternative implementation in this section that does the more "restful" thing of issuing a `DELETE` on `/users/<id>`.

As browsers only allow `GET` and `POST`, we need support from Spring Boot to turn the `POST` from the browser into a `DELETE`. This support comes in the form of the `HiddenHttpMethodFilter` class and can easily be enabled by setting the `spring.mvc.hiddenmethod.filter.enabled` property in the `application.properties` file:

`src/main/resources/application.properties`

```
spring.mvc.hiddenmethod.filter.enabled=true
```

We can update `UserController` to map our `doDeleteMethod` on the `DELETE` HTTP method:

`com.tamingthymeleaf.application.user.web.UserController`

```
@DeleteMapping("/{id}") ①
public String doDeleteUser(@PathVariable("id") UserId userId) {
    service.deleteUser(userId);

    return "redirect:/users";
}
```

① Use `@DeleteMapping` instead of `@PostMapping`.

Update the `deleteUrl` in the data cell :

`src/main/resources/templates/users/list.html`

```
<td th:x-data="|{name: '${user.userName.fullName}', deleteUrl:
'/users/${user.id.asString()}' }|"
    class="px-6 py-4 whitespace-no-wrap text-right text-sm font-medium">
    <a href="javascript:void(0)"
        class="text-green-600 hover:text-green-900"
        @click="$dispatch('open-delete-modal', {name, deleteUrl})">
        Delete</a>
    </td>
```

Add `th:method="delete"` on the form:

`src/main/resources/templates/users/list.html`

```
<form id="deleteForm" enctype="multipart/form-data" th:method="delete"
      x-bind:action="getDeleteUrl()">
```

Thymeleaf will render the `<form>` with a `method="post"` and also include a hidden input that has the actual HTTP method we want to execute.

This is what the browser will render:

```
<form id="deleteForm" enctype="multipart/form-data" method="post" x-
bind:action="getDeleteUrl()" action="/users/848f78a6-cd1e-417e-a423-
6a5ff8aabc75">
    <input type="hidden" name="_method" value="delete">
    ...
```

If we now restart everything, we will see that the delete also works fine this way.



*Which option to choose?*

It does not really matter if you use the dedicated URL or the `DELETE` mapping. Use what makes most sense to you.

Just one thing to be careful about is that Spring Boot made the [HiddenHttpMethodFilter](#) opt-in after some bug reports ([#16953](#), [#18088](#)) when the filter was used. If you have a similar use-case as the reports, you might want to avoid the filter.

## 13.3. Flash attributes

The delete works nicely, but the usability could be improved. The modal just closes without any indication that the user was actually removed. It would be a lot better to show a message after the redirect to the list of users. Something like "User Francine Connely was deleted."

Spring MVC has the concept of *flash attributes*. These attributes can be added in the `POST` method of the controller and will be made available in the `GET` method following the redirect we do at the end of the `@PostMapping` method.

To add flash attributes, we need to inject an instance of the `org.springframework.web.servlet.mvc.support.RedirectAttributes` interface in our controller method:

`com.tamingthymeleaf.application.user.web.UserController`

```

@PostMapping("/{id}/delete")
public String doDeleteUser(@PathVariable("id") UserId userId,
                           RedirectAttributes redirectAttributes) {
    ①
    User user = service.getUser(userId)
        .orElseThrow(() -> new UserNotFoundException
    (userId)); ②

    service.deleteUser(userId);

    redirectAttributes.addFlashAttribute("deletedUserName",
                                         user.getUserName
                                         ().getFullName()); ③

    return "redirect:/users";
}

```

- ① Add `RedirectAttributes` as a parameter to the method. Spring will automatically inject a proper instance at runtime.
- ② Retrieve the user.
- ③ Store the full name of the user under the `deletedUserName` key as a flash attribute.

Next step is adding a new Thymeleaf fragment `alerts.html` for the success message. It contains of 2 parts:

- The first part is a `<div>` that has the HTML for the alert message, taking a single argument being the String that should be displayed.

- The second part is a bit of JavaScript to make the close button on the alert message work.

This is the full listing:

`src/main/resources/templates/fragments/alerts.html`

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org"
      lang="en">
<div th:fragment="success(message)"
      class="max-w-7xl mx-auto px-4 sm:px-6 md:px-8 my-3"
      x-data="successMessageAlert()"
      x-show="isAlertVisible()"> ①
<div class="rounded-md bg-green-50 p-4">
    <div class="flex">
        <div class="flex-shrink-0">
            <!-- Heroicon name: solid/check-circle -->
            <svg class="h-5 w-5 text-green-400"
                  xmlns="http://www.w3.org/2000/svg" viewBox="0 0 20 20"
                  fill="currentColor" aria-hidden="true">
                <path fill-rule="evenodd"
                      d="M10 18a8 8 0 100-16 8 8 0 000 16zm3.707-9.293a1 1 0 00-1.414-1.414L9 10.586 7.707 9.293a1 1 0 00-1.414 1.414l2
2a1 1 0 001.414 0l4-4z"
                      clip-rule="evenodd"/>
            </svg>
        </div>
        <div class="ml-3">
            <p class="text-sm font-medium text-green-800"
               th:text="${message}"> ②
                Successfully uploaded
            </p>
        </div>
        <div class="ml-auto pl-3">
            <div class="-mx-1.5 -my-1.5">
                <button class="inline-flex bg-green-50 rounded-md p-1.5 text-green-500 hover:bg-green-100 focus:outline-none focus:ring-2 focus:ring-offset-2 focus:ring-green-50 focus:ring-green-600"
                       @click="hideAlert"> ③
                    <span class="sr-only">Dismiss</span>
                    <!-- Heroicon name: solid/x -->
                    <svg class="h-5 w-5"
                          xmlns="http://www.w3.org/2000/svg" viewBox="0 0 20 20"
                          fill="currentColor"
                          aria-hidden="true">
                        <path fill-rule="evenodd"
                              d="M10 18a8 8 0 100-16 8 8 0 000 16zm3.707-9.293a1 1 0 00-1.414-1.414L9 10.586 7.707 9.293a1 1 0 00-1.414 1.414l2
2a1 1 0 001.414 0l4-4z"
                              clip-rule="evenodd"/>
                    </svg>
                </button>
            </div>
        </div>
    </div>
</div>
```

```

d="M4.293 4.293a1 1 0 011.414 0L10
8.586l4.293-4.293a1 1 0 111.414 1.414L11.414 10l4.293 4.293a1 1 0 01-
1.414 1.414L10 11.414l-4.293 4.293a1 1 0 01-1.414-1.414L8.586 10 4.293
5.707a1 1 0 010-1.414z"
clip-rule="evenodd"/>
</svg>
</button>
</div>
</div>
</div>
</div>
</div>
<script th:fragment="success-js"> ④
function successMessageAlert() {
    return {
        show: true,
        isAlertVisible() {
            return this.show === true;
        },
        hideAlert() {
            this.show = false;
        }
    };
}
</script>

```

- ① Define a `success` fragment with the `x-data` and `x-show` attributes for the JavaScript.
- ② Use the `message` parameter to show the actual message.
- ③ Add a click listener to remove the alert when the user clicks on the x icon to close it.
- ④ Define a `success-js` fragment containing the relevant JavaScript.

We can now use the alert in `list.html`:

`src/main/resources/templates/users/list.html`

```

<div layout:fragment="page-content">
    ...
    <div th:if="${deletedUserName}"> ①
        <div th:replace="fragments/alerts ::"
success(${user.delete.success(${deletedUserName})})"></div> ②
    </div>
</div>

```

- ① Use `th:if` to only render the alert when the `deletedUserName` flash attribute is present
- ② Use the value of `deletedUserName` to display the message.

Update `messages.properties` with the translation:

`src/main/resources/i18n/messages.properties`

```
user.delete.success=User {0} was deleted successfully.
```

Also include the JavaScript part:

`src/main/resources/templates/users/list.html`

```
<th:block layout:fragment="page-scripts">
  ...
  <script th:replace="fragments/alerts :: success-js"></script>
</th:block>
```

The result looks like this in the browser:

NAME	GENDER	BIRTHDAY	EMAIL	Actions
Jeremiah Bailey	FEMALE	2003-09-19	jeremiah.bailey@gmail.com	Edit
Chun Dibbert	FEMALE	1996-10-27	chun.dibbert@gmail.com	Edit
Jasper Erdman	MALE	1988-05-12	jasper.erdman@hotmail.com	Edit
Starr Gorczany	FEMALE	1997-09-25	starr.gorczany@yahoo.com	Edit
Hilaria Haag	FEMALE	1995-06-08	hilaria.haag@yahoo.com	Edit
Ollie Hammes	MALE	1991-08-16	ollie.hammes@gmail.com	Edit
Roderick Heidenreich	MALE	2009-10-22	roderick.heidenreich@hotmail.com	Edit
Thu Heller	MALE	2006-06-11	thu.heller@gmail.com	Edit
Meri Jaskolski	MALE	1984-11-09	meri.jaskolski@hotmail.com	Edit

Figure 63. Alert message confirming that the delete was a success

If you now refresh the user list page manually, the green alert message will no longer be there. This is because the flash attribute is automatically removed after it was used during the redirect.

Due to the JavaScript we added, the user can also close the alert with the cross icon on the right side of the alert.

## 13.4. Summary

In this chapter, you learned:

- How to delete an entity using a dedicated URL.
- How to delete an entity using the [HiddenHttpMethodFilter](#).
- How to use AlpineJS to create a modal dialog.
- How to add a confirmation message upon redirect using flash attributes.

# Chapter 14. Security

Any non-trivial application will require some form of security. The Spring Security project has a lot of modules that will make it easier to build secure websites. This chapter will certainly not handle everything about Spring Security, but will serve as a good introduction to get started with it.

Security consists of 2 major parts:

- Authentication: Determine if a user is who the user claims he or she is. In simple terms, this is the part that verifies if the username and password match. That way, we can be sure users are really who they claim they are.
- Authorization: Once we know who the user is, we need to determine its access rights. What part of the application can the user access and what part is hidden or forbidden for them?

## 14.1. Default Spring Security

Adding Spring Security to the project is trivial, but it can take some effort to fully configure it.

To get started, we will add the dependency on Spring Security and the corresponding test utilities to the `pom.xml`:

```
<dependencies>
    ...
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
    </dependency>
    ...
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

That's it. We are done (Not quite 😊)!

Joking aside, we do have an application that is secured now. By default, there is a single user with username `user` and an auto-generated password. If you run the application and check the console output, you should see something like this:

**Using generated security password: 8897243f-ff75-4683-8ae3-8247d0282f22**

Access the application on <http://localhost:8080> and you will now be greeted with a login form:

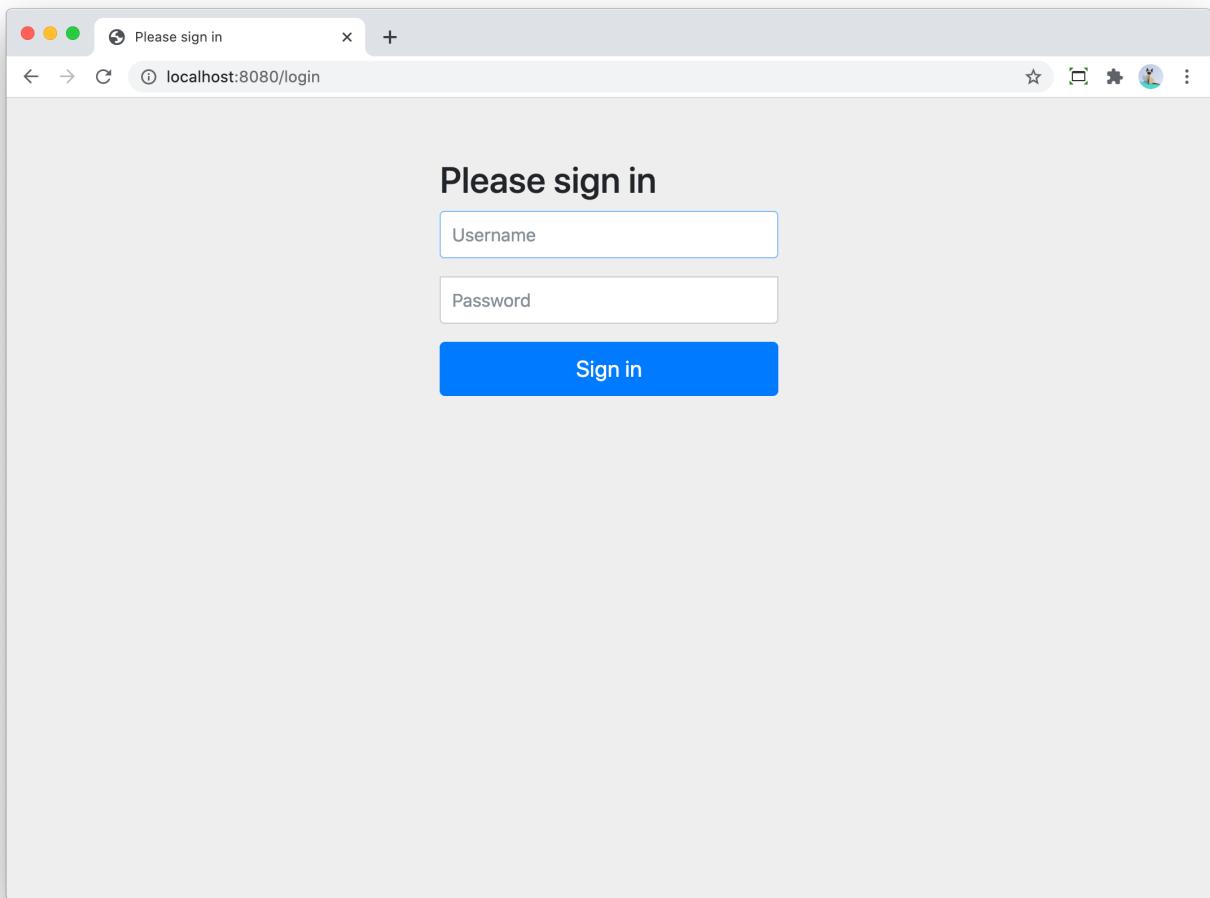


Figure 64. Default Spring Security login form

When the password is wrong, an error message is shown:

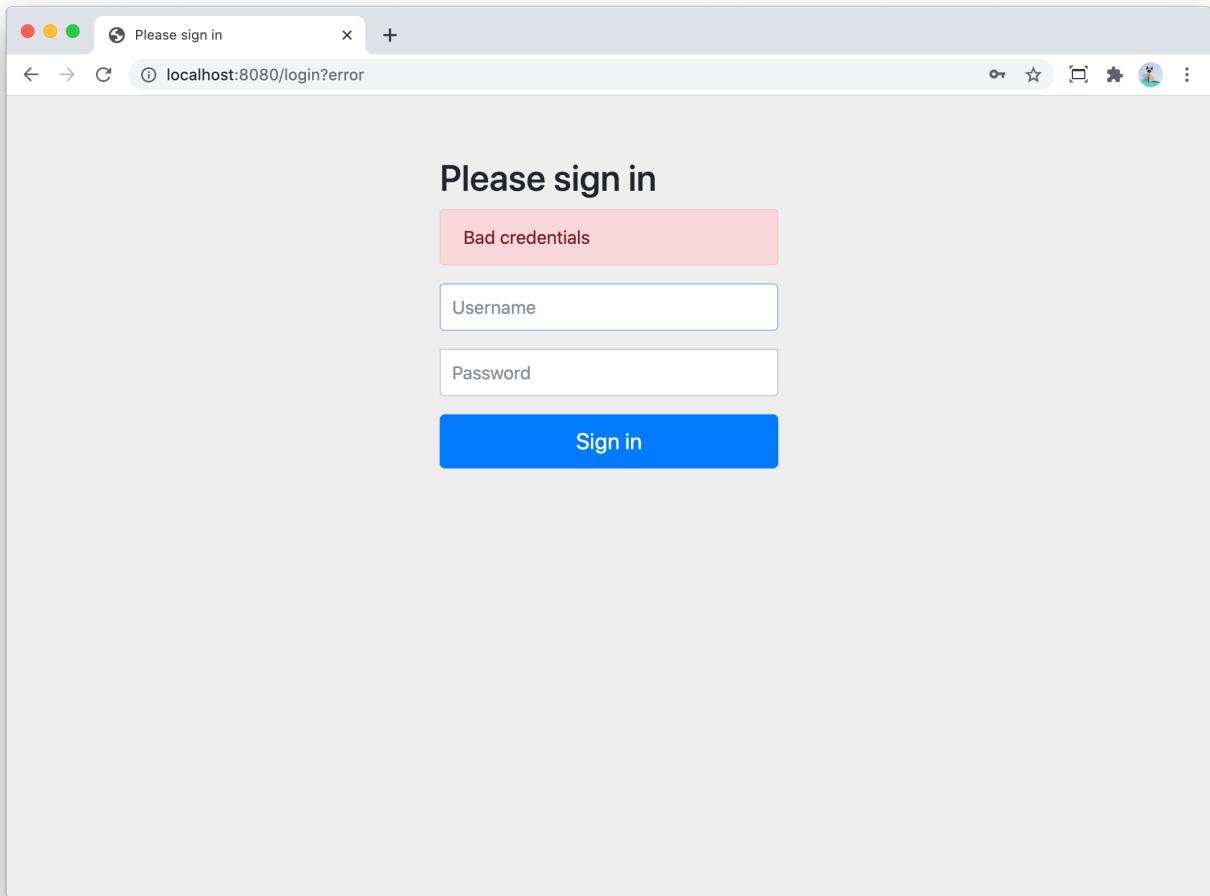


Figure 65. Typing a wrong password shows an error message

Use [user](#) and the auto-generated password and you will see the application again.

If you manually enter <http://localhost:8080/logout> after the login was ok, then you get a confirmation request:

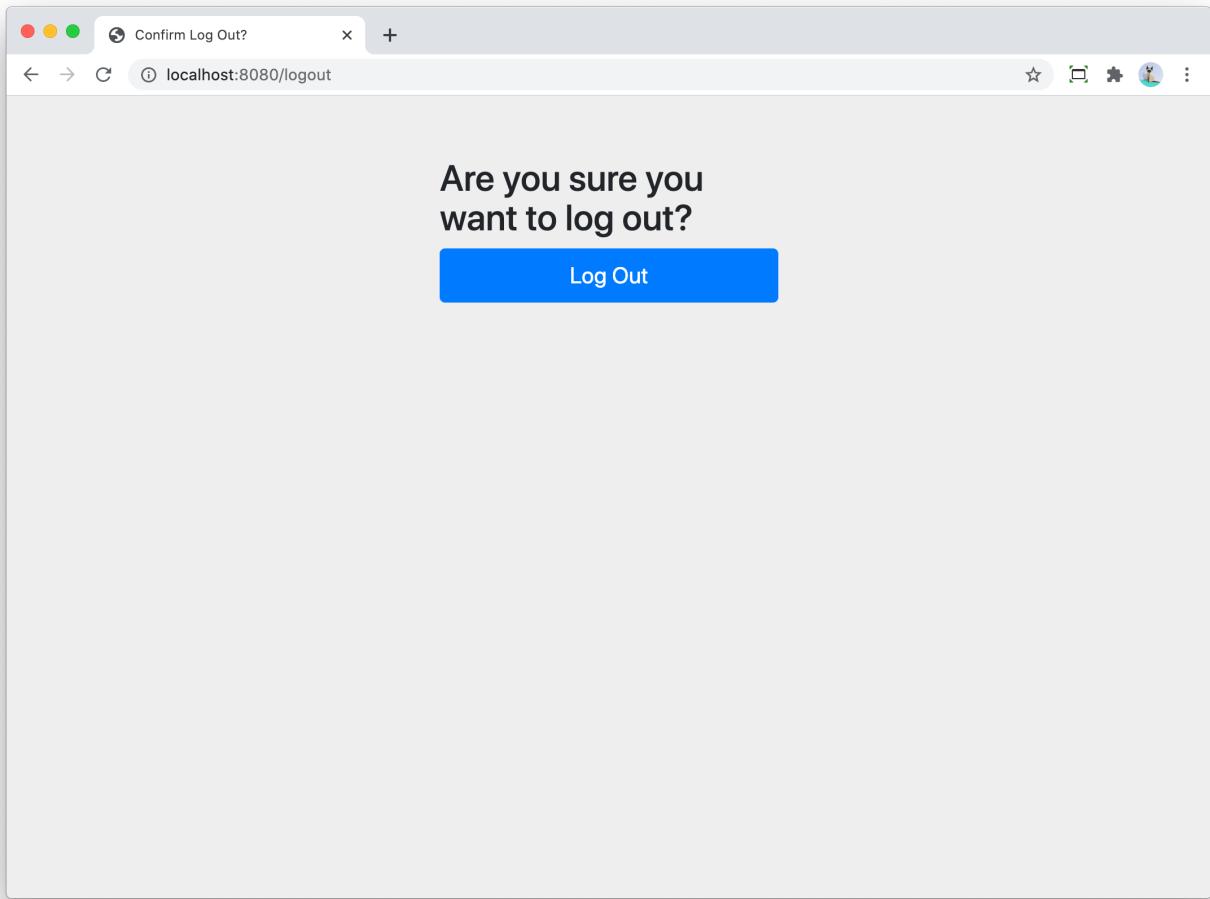


Figure 66. Default confirmation request from Spring Security to log out

After log out, you end up back on the login form:

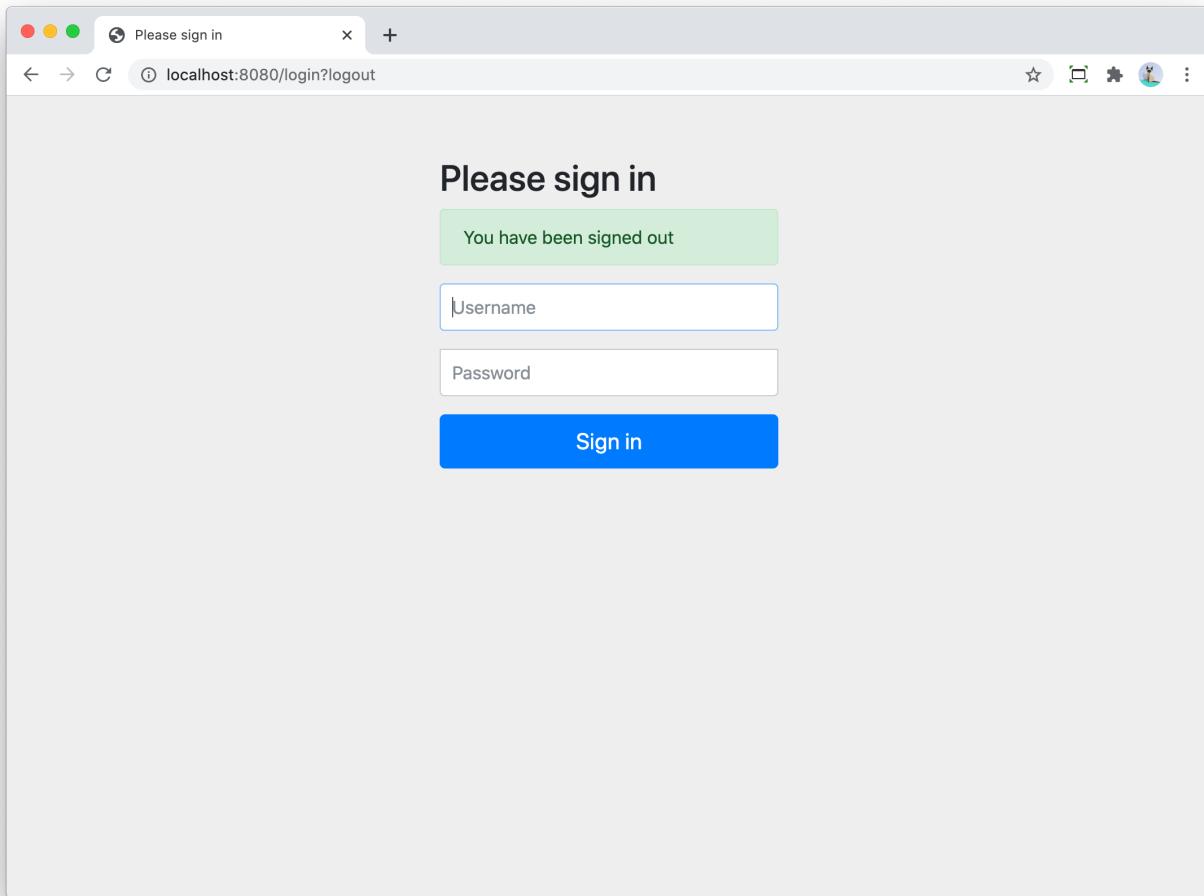


Figure 67. Default login form after successful log out

So, while this works with a minimal effort, there are quite some things missing:

- We only have a single user. The security users and the users in our application are not the same.
- The login form is not styled according to our application.
- We cannot choose the password for the users. On top of that, it is different for each run of the application.

## 14.2. Hardcoded password

We will apply customizations to the configuration bit by bit throughout this chapter. For starters, we will define a fixed password for the single user.

The heart of security configuration always starts from an `@Configuration` class that extends `org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter`. I usually place this in the `infrastructure.security` package and name it `WebSecurityConfiguration`:

```
package com.tamingthymeleaf.application.infrastructure.security;

import org.springframework.context.annotation.Configuration;
import
```

```

org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.crypto.password.PasswordEncoder;

@Configuration ①
public class WebSecurityConfiguration extends
WebSecurityConfigurerAdapter { ②

    private final PasswordEncoder passwordEncoder;

    public WebSecurityConfiguration(PasswordEncoder passwordEncoder) {
③
        this.passwordEncoder = passwordEncoder;
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws
Exception { ④
        auth.inMemoryAuthentication() ⑤
            .withUser("user") ⑥
            .password(passwordEncoder.encode("verysecure")) ⑦
            .roles("USER"); ⑧
    }
}

```

- ① Annotate the class with `@Configuration` so the component scanning will pick it up automatically.
- ② Extend from `WebSecurityConfigurerAdapter` to be able to override the Spring Security defaults.
- ③ Inject the `PasswordEncoder` instance to be able to securely encode the password of the user. It has little use here since the password is in plain text in the source code, but this is something we will definitely need when we want to store passwords in the database securely.
- ④ Override the `configure(AuthenticationManagerBuilder)` method to configure the security users.
- ⑤ Use the builder to add in memory users.
- ⑥ Add a user with username `user`.
- ⑦ Set the password of the user to `verysecure`.
- ⑧ Give the account the role of `USER`.

There is no `PasswordEncoder` created by Spring Boot by default, so we need to create such a bean ourselves in `TamingThymeleafApplicationConfiguration`:

```
@Bean
public PasswordEncoder passwordEncoder() {
    return PasswordEncoderFactories.
createDelegatingPasswordEncoder();
}
```

We use the `DelegatingPasswordEncoder` here which contains all the known encoding schemes that Spring Security has. The advantage of using this is that the password is prefixed with the used encoding scheme when stored in the database.

It would look something like this:

```
{bcrypt}$2a$10$NjRCquznza.Q2CDHwSgTu.U6WfEYLg3s0UWudKRGK3G..A7uK9iLm
```

This allows us to use different encoding schemes depending on the needed security and allows migrating to more secure schemes if the need would arise.

If we now restart the application, we should be able to log on using `user/verysecure`.

## 14.3. User roles

### 14.3.1. URL based authorization

We have used the single role `USER` so far, but most applications have multiple roles for their users. This allows to only allow certain operations (e.g. to delete a user) for certain roles (e.g. administrators).

As an example of how this works, we will create a second hardcoded user `admin` which has the `USER` and `ADMIN` roles:

`com.tamingthymeleaf.application.infrastructure.security.WebSecurityConfiguration`

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws
Exception {
    auth.inMemoryAuthentication()
        .withUser("user")
        .password(passwordEncoder.encode("verysecure"))
        .roles("USER")
        .and()
        .withUser("admin")
        .password(passwordEncoder.encode("evenmoresecure"))
        .roles("USER", "ADMIN");
}
```

We can now override the `configure(HttpSecurity http)` method to determine what user role is allowed to access what part of the application:

`com.tamingthymeleaf.application.infrastructure.security.WebSecurityConfiguration`

```

@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests() ①
        .antMatchers("/users/create").hasRole("ADMIN") ②
        .antMatchers("/users/*/delete").hasRole("ADMIN") ③
        .antMatchers(HttpMethod.GET, "/users/*").hasRole("USER") ④
        .antMatchers(HttpMethod.POST, "/users/*").hasRole("ADMIN") ⑤
        .and()
        .formLogin().permitAll() ⑥
        .and()
        .logout().permitAll(); ⑦
}

```

- ① We want requests to be authorized.
- ② Only a user with **ADMIN** role can access `/users/create`. This is valid for any HTTP method (so `GET`, `POST`, ...)
- ③ Only a user with **ADMIN** role can access a URL that matches with `/users/*/delete`. The `*` means any character except `/`.
- ④ We can also secure a path with a specific HTTP method. Here we state that a user in the **USER** role can only do a `GET` on any sub-path of `/users`.
- ⑤ Using that same method, we allow **ADMIN** users to do a `POST` on those sub-paths.
- ⑥ We want the default form login. Any user (authenticated or not) should be able to access the login form.
- ⑦ Provide default logout support and allow everybody access. This will make Spring Security add a handler for a `POST` on `/logout` to log out the current logged on user.

With this configuration in place, the `admin` user will be able to do everything. The `user` will no longer be able to create users, edit users or delete users.

After login with `user`, we still see the create button and the edit and delete links:

NAME	GENDER	BIRTHDAY	EMAIL	
Francine Connelly	FEMALE	1988-07-29	francine.connelly@gmail.com	<a href="#">Edit</a>
Lynn Daniel	FEMALE	2007-05-21	lynn.daniel@yahoo.com	<a href="#">Edit</a>
Gordon Farrell	FEMALE	1981-11-13	gordon.farrell@yahoo.com	<a href="#">Edit</a>
Kelley Grimes	MALE	2008-04-24	kelley.grimes@yahoo.com	<a href="#">Edit</a>
Irving Hill	MALE	1989-05-31	irving.hill@yahoo.com	<a href="#">Edit</a>
Cornell Jacobson	MALE	1987-01-17	cornell.jacobson@hotmail.com	<a href="#">Edit</a>
Shelby Johnston	MALE	1982-03-11	shelby.johnston@hotmail.com	<a href="#">Edit</a>
Zofia Lang	MALE	1985-10-04	zofia.lang@hotmail.com	<a href="#">Edit</a>
Haywood McCullough	FEMALE	1996-01-22	haywood.mccullough@gmail.com	<a href="#">Edit</a>
Teri Moen	MALE	2008-08-16	teri.moen@yahoo.com	<a href="#">Edit</a>

This is normal as we did not specify anything in our Thymeleaf templates that some parts should be made invisible for certain roles.

As soon as we try something (E.g. the 'Create user' button), we get a 403 FORBIDDEN response:

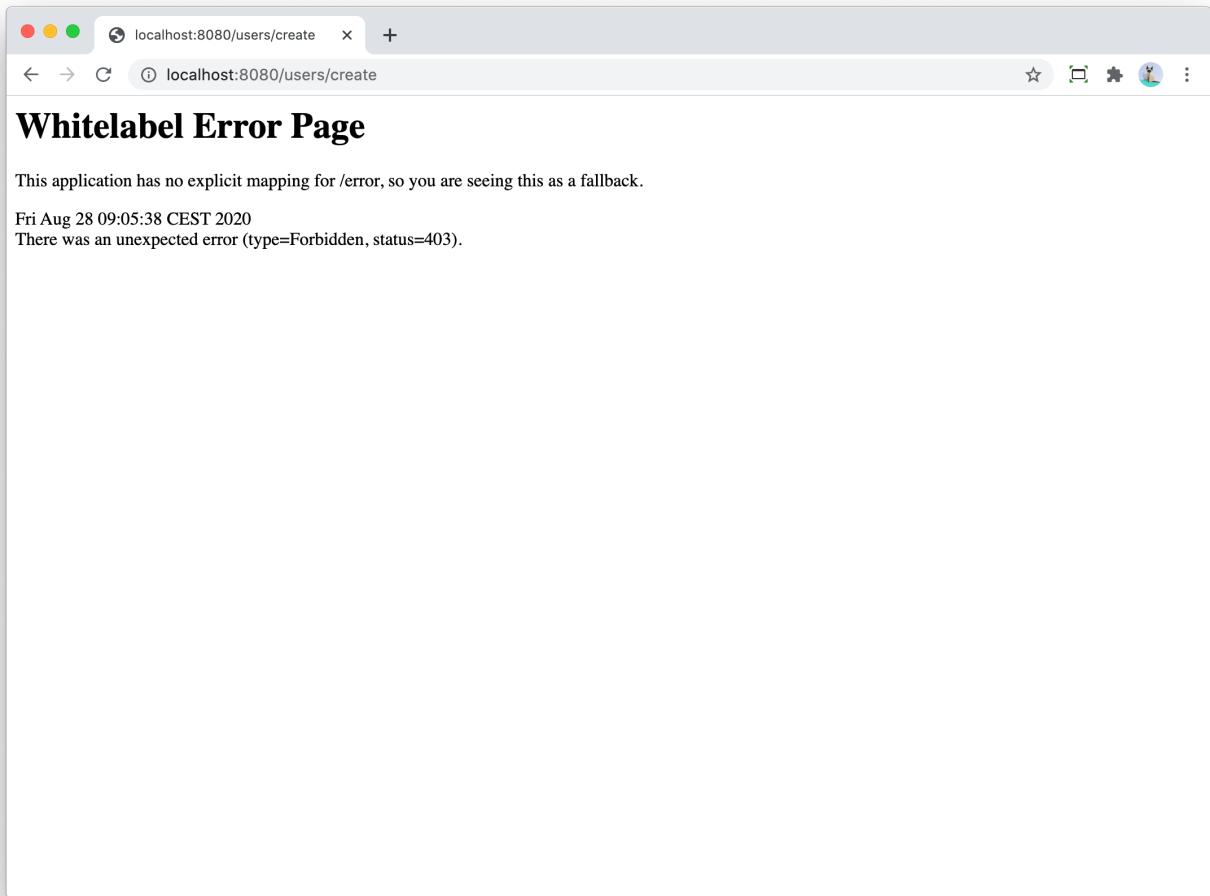


Figure 68. Whitelabel error page for 403 FORBIDDEN errors

We can make this nicer using the mechanism we discussed at [Custom error pages](#).

Create `templates/error/403.html`:

```

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
      layout:decorate="~{layout/layout}"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      lang="en">
<head>
    <title th:text="#{error}">Error</title>
</head>
<body>
<!--/*@thymesVar id="url" type="String"-->
<div layout:fragment="page-content">
    <div class="max-w-7xl mx-auto px-4 sm:px-6 md:px-8">
        <div class="py-4">
            <div class="text-gray-500">
                <p th:text="#{error.page.forbidden}" class="mb-6">The
current user is not allowed to access this page.</p>

```

```
<a th:href="@{/}" class="flex items-center text-sm text-green-600 hover:text-green-900">
    <svg viewBox="0 0 20 20" fill="currentColor"
class="w-4 h-4 mr-2">
        <path fill-rule="evenodd"
            d="M9.707 16.707a1 1 0 01-1.414 0l-6-6a1 1
0 01-1.414l6-6a1 1 0 011.414 1.414L5.414 9H17a1 1 0 110 2H5.414l4.293
4.293a1 1 0 010 1.414z"
            clip-rule="evenodd"></path>
    </svg>
    <span th:text="#{back.to.home.page}">Back to home
page</span> </a>
</div>
</div>
</div>
</body>
</html>
```

Trying to access `/users/create` with `user` will now show a nicer error page:

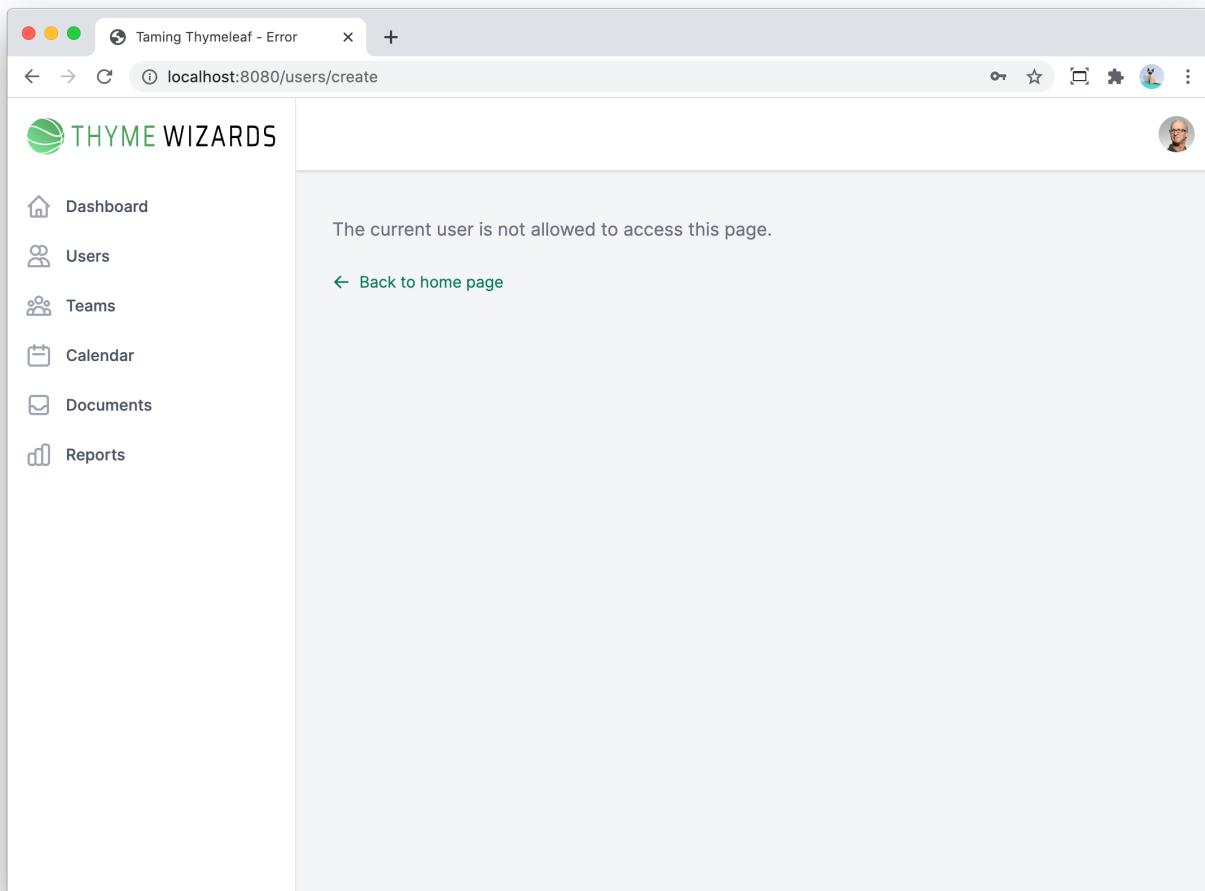


Figure 69. Custom error page for 403 FORBIDDEN errors

### 14.3.2. Annotation based authorization

We configured authorization centrally in our `WebSecurityConfiguration` class in the previous section. There is an alternative way of configuration that uses an annotation on the controller methods.

We need to first enable this on `WebSecurityConfiguration` via the `org.springframework.security.config.annotation.method.configuration.EnableGlobalMethodSecurity` annotation:

```
@Configuration
@EnableGlobalMethodSecurity(securedEnabled = true)
public class WebSecurityConfiguration extends
WebSecurityConfigurerAdapter {
```

Adding this annotation enables the use of specific annotations on the controller methods. Using `securedEnabled=true` allows us to use `org.springframework.security.access.annotation.Secured` which is a Spring specific annotation.

If you want to use the JSR-250 annotation `javax.annotation.security.RolesAllowed` instead, then use `@EnabledGlobalMethodSecurity` like this:



```
@Configuration
@EnableGlobalMethodSecurity(jsr250Enabled = true)
public class WebSecurityConfiguration extends
WebSecurityConfigurerAdapter {
```

With this in place, we can now add the `@Secured` annotation with the name of the role to secure certain URLs:

```
@GetMapping("/create")
@Secured("ROLE_ADMIN") ①
public String createUserForm(Model model) {
    model.addAttribute("user", new CreateUserFormData());
    model.addAttribute("genders", List.of(Gender.MALE, Gender.
FEMALE, Gender.OTHER));
    model.addAttribute("EditMode", EditMode.CREATE);
    return "users/edit";
}
```

① The `@Secured` annotation will ensure that only users with the `ADMIN` role will be able to access the `/users/create` URL.



Be careful: in the security configuration, we use:

```
...
.roles("USER", "ADMIN")
```

But in the annotation, we use a `ROLE_` prefix:

```
@Secured("ROLE_ADMIN")
```

After adding `@Secured("ROLE_ADMIN")` to all appropriate places in the `UserController` class, we get the exact same behaviour as before.

## 14.4. Thymeleaf integration

### 14.4.1. User specific views

To improve the user experience, we should hide any buttons, links or other content that the logged on user is not supposed to see. To do that, we can use the Thymeleaf Spring Security integration.

Start by adding the needed dependency in the `pom.xml`:

```
<dependencies>
  ...
  <dependency>
    <groupId>org.thymeleaf.extras</groupId>
    <artifactId>thymeleaf-extras-springsecurity5</artifactId>
  </dependency>
  ...
</dependencies>
```

We can now edit `templates/users/list.html` to take the current user role into account when rendering the template:

`src/main/resources/templates/users/list.html`

```
<div th:replace="fragments/titles :: title-with-button(${users.title},
'user-add', ${users.add}, @{/users/create})"
sec:authorize="hasRole('ADMIN')"></div>
```

By adding the `sec:authorize` attribute, we control the rendering of the `<div>` based on the expression we give to the attribute. In this example, only users with the `ADMIN` role will view the 'Create user' button.

We can do something similar for the 'Edit' and 'Delete' links. First, hide the table header for non-admin users:

src/main/resources/templates/users/list.html

```
<tr>
    <th th:replace="fragments/table :: header(#{user.name})"></th>
    <th th:replace="fragments/table :: header(title=#{user.gender},hideOnMobile=true)"></th>
    <th th:replace="fragments/table :: header(title=#{user.birthday},hideOnMobile=true)"></th>
    <th th:replace="fragments/table :: header(title=#{user.email},hideOnMobile=true)"></th>
    <th:block sec:authorize="hasRole('ADMIN')"> ①
        <th th:replace="fragments/table :: header('')"></th>
        <th th:replace="fragments/table :: header('')"></th>
    </th:block>
</tr>
```

Do the same for the actual data in the table:

src/main/resources/templates/users/list.html

```
<tr class="bg-white" th:each="user : ${users}">
    <td th:replace="fragments/table :: data(contents=${user.userName.fullName},primary=true)"></td>
    <td th:replace="fragments/table :: data(contents=${user.gender},hideOnMobile=true)"></td>
    <td th:replace="fragments/table :: data(contents=${user.birthday},hideOnMobile=true)"></td>
    <td th:replace="fragments/table :: data(contents=${user.email.asString()},hideOnMobile=true)"></td>
    <th:block sec:authorize="hasRole('ADMIN')"> ①
        <td th:replace="fragments/table :: dataWithLink('Edit', @{'/users/' + ${user.id.asString()}})"></td>
        <td th:x-data="|{name: '${user.userName.fullName}', deleteUrl: '/users/${user.id.asString()}/delete'}|"
            class="px-6 py-4 whitespace-no-wrap text-right text-sm font-medium">
            <a href="javascript:void(0)"
                class="text-green-600 hover:text-green-900"
                @click="$dispatch('open-delete-modal', {name, deleteUrl})">Delete</a>
        </td>
    </th:block>
</tr>
```

① Wrap the 'Edit' and 'Delete' links with a `<th:block>` that has the `sec:authorize` attribute to indicate the role that can view the links.

The XML namespace to use for `sec` is <http://www.thymeleaf.org/extras/spring-security>:



```
<html
    xmlns:th="http://www.thymeleaf.org"
    xmlns:sec="http://www.thymeleaf.org/extras/spring-
security"

    xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
    layout:decorate="~{layout/layout}"
    th:with="activeMenuItem='users'>
```

Rendered by the browser, we get the following result as `user`:

NAME	GENDER	BIRTHDAY	EMAIL
Francine Connelly	FEMALE	1988-07-29	francine.connelly@gmail.com
Lynn Daniel	FEMALE	2007-05-21	lynn.daniel@yahoo.com
Gordon Farrell	FEMALE	1981-11-13	gordon.farrell@yahoo.com
Kelley Grimes	MALE	2008-04-24	kelley.grimes@yahoo.com
Irving Hill	MALE	1989-05-31	irving.hill@yahoo.com
Cornell Jacobson	MALE	1987-01-17	cornell.jacobson@hotmail.com
Shelby Johnston	MALE	1982-03-11	shelby.johnston@hotmail.com
Zofia Lang	MALE	1985-10-04	zofia.lang@hotmail.com
Haywood McCullough	FEMALE	1996-01-22	haywood.mccullough@gmail.com
Teri Moen	MALE	2008-08-16	teri.moen@yahoo.com

Figure 70. Create button and edit links are gone when logging on with `user`

#### 14.4.2. Current logged on user information

Another nice feature of the Thymeleaf Spring Security integration is displaying information about the logged on user.

We can use this to show the username of the current user for example.

We can update `templates/fragments/top-menu.html` by adding this:

```
<div class="block px-4 py-2 text-sm text-gray-700 font-mono border-b"
    sec:authentication="name"></div> ①
<div class="block px-4 py-2 text-sm text-gray-700 font-mono border-b"
    sec:authentication="principal.authorities"></div> ②
```

- ① Show the username of the logged on user
- ② Show the list of roles of the logged on user

The user popup menu now shows the name and roles of the current logged on user:

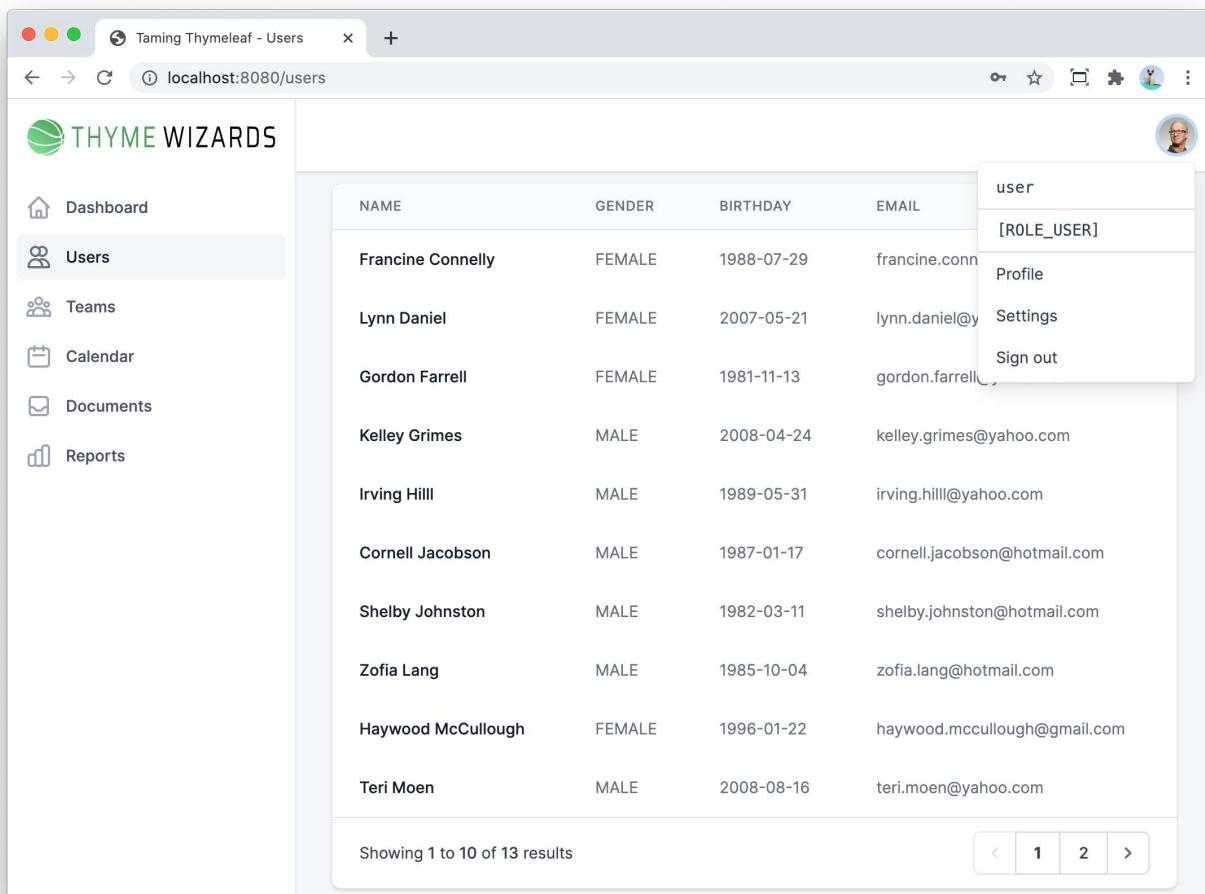


Figure 71. Popup menu for user

The screenshot shows a web browser window titled "Taming Thymeleaf - Users" at the URL "localhost:8080/users". The left sidebar has a "THYME WIZARDS" logo and links for Dashboard, Users (which is selected), Teams, Calendar, Documents, and Reports. The main content area is titled "Users" and displays a table of users with columns: NAME, GENDER, BIRTHDAY, and EMAIL. The users listed are Francine Connelly, Lynn Daniel, Gordon Farrell, Kelley Grimes, Irving Hill, Cornell Jacobson, Shelby Johnston, Zofia Lang, Haywood McCullough, and Teri Moen. Each user row has an "Edit" link on the right. A context menu is open over the first user, showing options: "admin", "[ROLE\_ADMIN, ROLE\_USER]", "Profile", "Settings", and "Sign out".

NAME	GENDER	BIRTHDAY	EMAIL	
Francine Connelly	FEMALE	1988-07-29	francine.connelly@yahoo.com	<a href="#">Edit</a>
Lynn Daniel	FEMALE	2007-05-21	lynn.daniel@yahoo.com	<a href="#">Edit</a>
Gordon Farrell	FEMALE	1981-11-13	gordon.farrell@yahoo.com	<a href="#">Edit</a>
Kelley Grimes	MALE	2008-04-24	kelley.grimes@yahoo.com	<a href="#">Edit</a>
Irving Hill	MALE	1989-05-31	irving.hill@yahoo.com	<a href="#">Edit</a>
Cornell Jacobson	MALE	1987-01-17	cornell.jacobson@hotmail.com	<a href="#">Edit</a>
Shelby Johnston	MALE	1982-03-11	shelby.johnston@hotmail.com	<a href="#">Edit</a>
Zofia Lang	MALE	1985-10-04	zofia.lang@hotmail.com	<a href="#">Edit</a>
Haywood McCullough	FEMALE	1996-01-22	haywood.mccullough@gmail.com	<a href="#">Edit</a>
Teri Moen	MALE	2008-08-16	teri.moен@yahoo.com	<a href="#">Edit</a>

Figure 72. Popup menu for admin

There are some other more advanced things that are possible. See [Thymeleaf - Spring Security integration modules](#) for more information.

## 14.5. Custom logon page

We have been using the default Spring Security logon page until now, but it is unlikely you will use that in your application. This section will show you how to create your own login page, so you can create one fitting the style of the application.

The following steps are needed to do this:

- Design a login page and write a Thymeleaf template to match that design. The page will need a `<form>` to submit the username and password with 2 inputs that use `username` and `password` as names for the inputs.
- Create a controller that will return the template at `/login`.
- Update `WebSecurityConfiguration` to use the custom login page.

Let's start with the `login.html` 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 - Login</title>
    <meta http-equiv="X-UA-Compatible" content="IE=edge"/>
    <meta name="viewport" content="width=device-width, initial-
scale=1"/>

    <link rel="stylesheet" href="https://rsms.me/inter/inter.css">
    <link rel="stylesheet" th:href="@{/css/application.css}">
</head>
<body>
<div class="min-h-screen bg-gray-50 flex flex-col justify-center py-12
sm:px-6 lg:px-8">

    <div class="sm:mx-auto sm:w-full sm:max-w-md">
        <div class="bg-white py-8 px-4 shadow sm:rounded-lg sm:px-10">
            <div class="sm:mx-auto sm:w-full sm:max-w-md mb-6">
                
            </div>
            <form th:action="@{/login}" method="post"> ①
                <div>
                    <label for="username" class="block text-sm font-
medium text-gray-700"
                          th:text="#{login.username}">
                        Username
                    </label>
                    <div class="mt-1 rounded-md shadow-sm">
                        <input id="username"
                               class="appearance-none block w-full px-3
py-2 border border-gray-300 rounded-md shadow-sm placeholder-gray-400
focus:outline-none focus:ring-green-500 focus:border-green-500 sm:text-
sm"
                               type="text"
                               required
                               name="username"> ②
                    </div>
                </div>
            <div class="mt-6">
                <label for="password" class="block text-sm font-
medium leading-5 text-gray-700"

```

```
        th:text="#{login.password}">
        Password
    </label>
    <div class="mt-1 rounded-md shadow-sm">
        <input id="password"
               class="appearance-none block w-full px-3
py-2 border border-gray-300 rounded-md shadow-sm placeholder-gray-400
focus:outline-none focus:ring-green-500 focus:border-green-500 sm:text-
sm"
               type="password"
               required
               name="password"> ③
    </div>
</div>

<div class="mt-6 flex items-center justify-between">
    <div class="flex items-center">
        <input id="remember_me" type="checkbox"
               class="h-4 w-4 text-green-600 focus:ring-
green-500 border-gray-300 rounded">
        <label for="remember_me" class="ml-2 block text-
sm text-gray-900">
            Remember me
        </label>
    </div>
    <div class="text-sm leading-5">
        <a href="#" class="font-medium text-green-600 hover:text-
green-500">
            Forgot your password?
        </a>
    </div>
</div>

<div class="mt-6">
<span class="block w-full rounded-md shadow-sm">
    <button type="submit"
           class="w-full flex justify-center py-2 px-4 border
border-transparent rounded-md shadow-sm text-sm font-medium text-white
bg-green-600 hover:bg-green-700 focus:outline-none focus:ring-2
focus:ring-offset-2 focus:ring-green-500">
        Sign in
    </button>
</span>
```

```

        </div>
    </form>
</div>
</div>
</div>
</body>
</html>
```

- ① Set the action of the form to `/login` as this is what Spring Security expects.
- ② `<input>` with `username` as `name` attribute value.
- ③ `<input>` with `password` as `name` attribute value.

Next, create the `LoginController`:

```

package com.tamingthymeleaf.application.infrastructure.web;

import
org.springframework.security.core.annotation.AuthenticationPrincipal;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class LoginController {

    @GetMapping("/login") ①
    public String login(@AuthenticationPrincipal UserDetails
userDetails) { ②
        if (userDetails == null) { ③
            return "login"; ④
        } else {
            return "redirect:/"; ⑤
        }
    }
}
```

- ① The `POST` to `/login` is handled by Spring Security, but we have to implement the `GET`.
- ② We have Spring inject the details of the currently logged on user via the `@AuthenticationPrincipal`.
- ③ If `userDetails` is `null`, there is no logged on user and...
- ④ ...we show the `login.html` template.
- ⑤ If somebody manually uses the `/login` URL in the browser while already logged on, we redirect away from the login page. This makes for a nice UX to avoid showing the login page to an already logged on user.

As a last step, update [WebSecurityConfiguration](#):

```
@Override  
protected void configure(HttpSecurity http) throws Exception {  
    http.authorizeRequests()  
        .requestMatchers(PathRequest.toStaticResources()  
().atCommonLocations()).permitAll() ①  
            .antMatchers("/img/*").permitAll() ②  
            .anyRequest().authenticated()  
            .and()  
            .formLogin()  
            .loginPage("/login") ③  
            .permitAll()  
            .and()  
            .logout().permitAll();  
}
```

- ① Because we now need access to the CSS *before* we are logged on, we need to expose that. `PathRequest.toStaticResources().atCommonLocations()` is a convenience method to expose `/css/*`, `/js/*`, `/images/*`, `/webjars/*` and `favicon.ico`.
- ② Our images are located under `/img/`, so we also need to allow those paths to everybody.
- ③ This specifies the URL where the login page lives. So this needs to match with our `@GetMapping` in the [LoginController](#).

With this in place, we are now greeted with our custom login page:

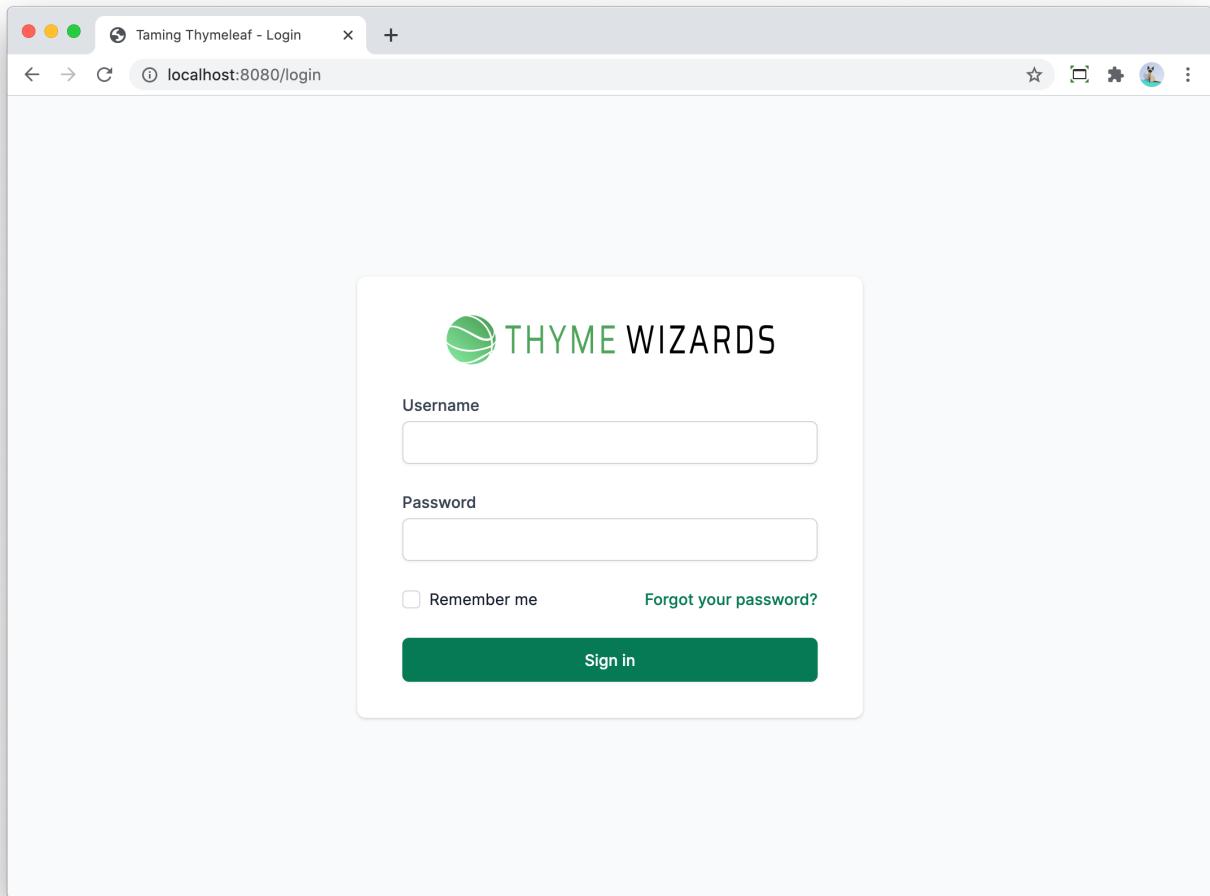


Figure 73. Custom login page

For convenience, let's make the logout link in the user menu also work.

Edit `templates/fragments/top-menu.html` and replace:

```
src/main/resources/templates/fragments/top-menu.html
```

```
<a href="#" class="block px-4 py-2 text-sm text-gray-700 hover:bg-gray-100"
    th:text="#{menu.signout}"
    role="menuitem">Sign out
</a>
```

with:

```
src/main/resources/templates/fragments/top-menu.html
```

```
<div class="block px-4 py-2 text-sm text-gray-700 hover:bg-gray-100">
    <form th:action="@{/logout}" method="post"> ①
        <button type="submit" th:text="#{menu.signout}">
            Sign out
        </button>
    </form>
```

```
</div>
```

- ① Add a `<form>` with `/logout` as the action. Spring Security will handle the `POST` to log out the current user and redirect back to the `/login` page.

When Spring Security redirects back to the login page after the logout, it adds a query parameter to the url of `logout`. We can use this to display a confirmation message about the logout.

`src/main/resources/templates/login.html`

```
<th:block th:if="${param.logout}"> ①
    <div th:replace="fragments/alerts ::"
success(message=#{login.logout.confirmation},useHorizontalPadding=false)
"></div>
</th:block>
```

- ① Query parameters are available in a Thymeleaf template under the `param` key.

For the alert message, we can re-use the `success` fragment from `alerts.html`. To make the alert look nice on the login form, we added a new `useHorizontalPadding` parameter to the fragment with a default value of `true`. This allows use to remove the horizontal padding dynamically since we don't need it here:

`src/main/resources/templates/fragments/alerts.html`

```
<div th:fragment="success(message)"
    class="max-w-7xl mx-auto my-3"
    th:with="useHorizontalPadding=${useHorizontalPadding?'true'}"
    th:classappend="${useHorizontalPadding?'px-4 sm:px-6 md:px-8':''}"
    x-data="successMessageAlert()"
    x-show="isAlertVisible()">
```

The result in the browser after log out:

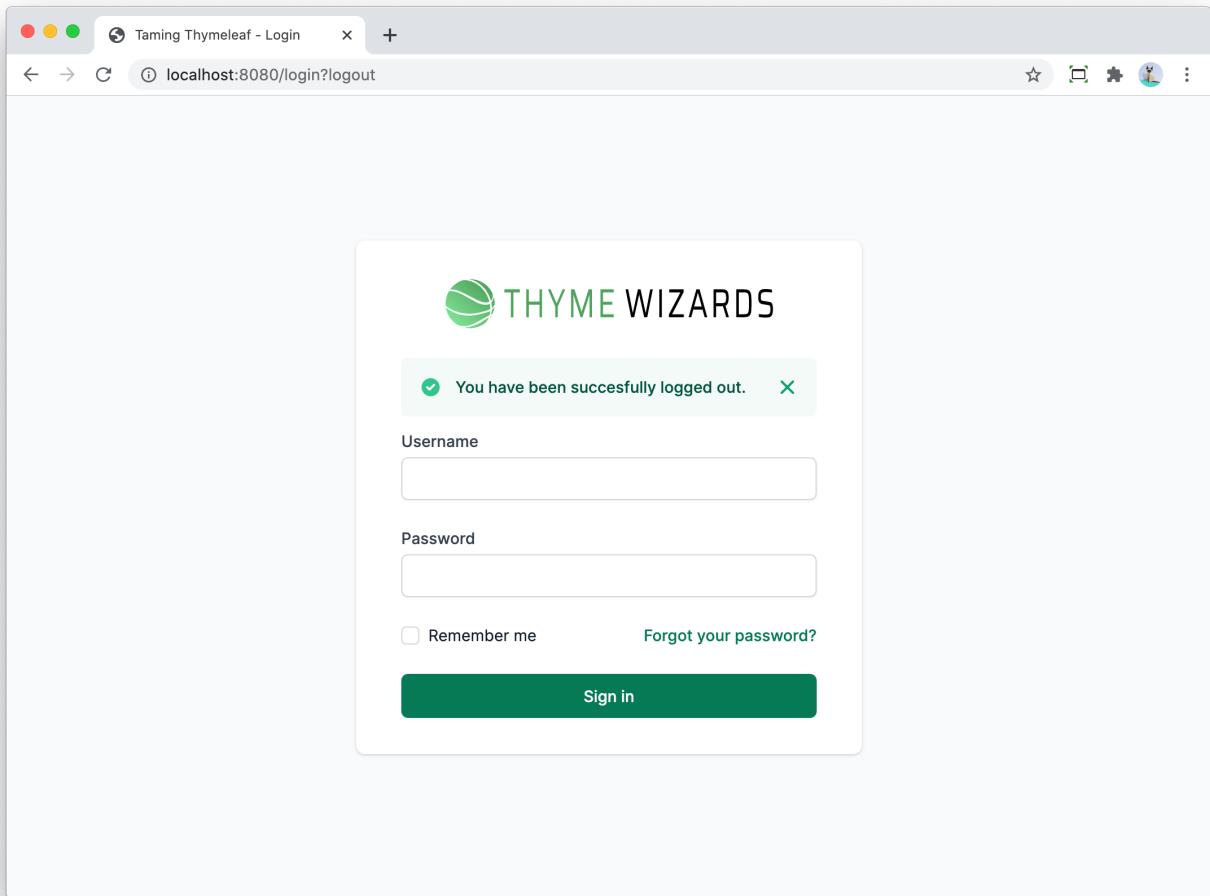


Figure 74. Custom login page after logout

The final part of our custom logon form is handling a login error. When the user was not found, or the password did not match, Spring Security will add a query parameter of `error` to the `/login` URL. We can use this to display an error message:

`src/main/resources/templates/login.html`

```
<th:block th:if="${param.error}"> ①
    <div th:replace="fragments/alerts ::"
        error(message=#{login.error},useHorizontalPadding=false)"></div>
</th:block>
```

We created a new fragment in `alerts.html` to make this work:

`src/main/resources/templates/fragments/alerts.html`

```
<div th:fragment="error(message)"
    class="max-w-7xl mx-auto my-3"
    th:with="useHorizontalPadding=${useHorizontalPadding?:'true'}"
    th:classappend="${useHorizontalPadding?'px-4 sm:px-6 md:px-8':''}"
    x-data="messageAlert()"
    x-show="isAlertVisible()">
```

```

<div class="rounded-md bg-red-50 p-4">
    <div class="flex">
        <div class="flex-shrink-0">
            <svg class="h-5 w-5 text-red-400"
                xmlns="http://www.w3.org/2000/svg" viewBox="0 0 20 20"
                fill="currentColor" aria-hidden="true">
                <path fill-rule="evenodd"
                    d="M10 18a8 8 0 100-16 8 8 0 000 16zM8.707
                    7.293a1 1 0 00-1.414 1.414L8.586 10l-1.293 1.293a1 1 0 101.414 1.414L10
                    11.414l1.293 1.293a1 1 0 001.414-1.414L11.414 10l1.293-1.293a1 1 0 00-
                    1.414-1.414L10 8.586 8.707 7.293z"
                    clip-rule="evenodd"/>
            </svg>
        </div>
        <div class="ml-3">
            <p class="text-sm font-medium text-red-800"
                th:text="${message}">
                Successfully uploaded
            </p>
        </div>
        <div class="ml-auto pl-3">
            <div class="-mx-1.5 -my-1.5">
                <button class="inline-flex rounded-md p-1.5 text-
                    red-500 hover:bg-red-100 focus:outline-none focus:bg-red-100 transition
                    ease-in-out duration-150"
                    @click="hideAlert">
                    <span class="sr-only">Dismiss</span>
                    <!-- Heroicon name: solid/x -->
                    <svg class="h-5 w-5"
                        xmlns="http://www.w3.org/2000/svg" viewBox="0 0 20 20"
                        fill="currentColor"
                        aria-hidden="true">
                        <path fill-rule="evenodd"
                            d="M4.293 4.293a1 1 0 011.414 0L10
                            8.586l4.293-4.293a1 1 0 101.414 1.414L11.414 10l4.293 4.293a1 1 0 01-
                            1.414 1.414L10 11.414l-4.293 4.293a1 1 0 01-1.414-1.414L8.586 10 4.293
                            5.707a1 1 0 010-1.414z"
                            clip-rule="evenodd"/>
                </svg>
            </button>
        </div>
    </div>
</div>
</div>

```

```
</div>
```

Try it out. Enter a wrong username and/or password and the browser should show this:

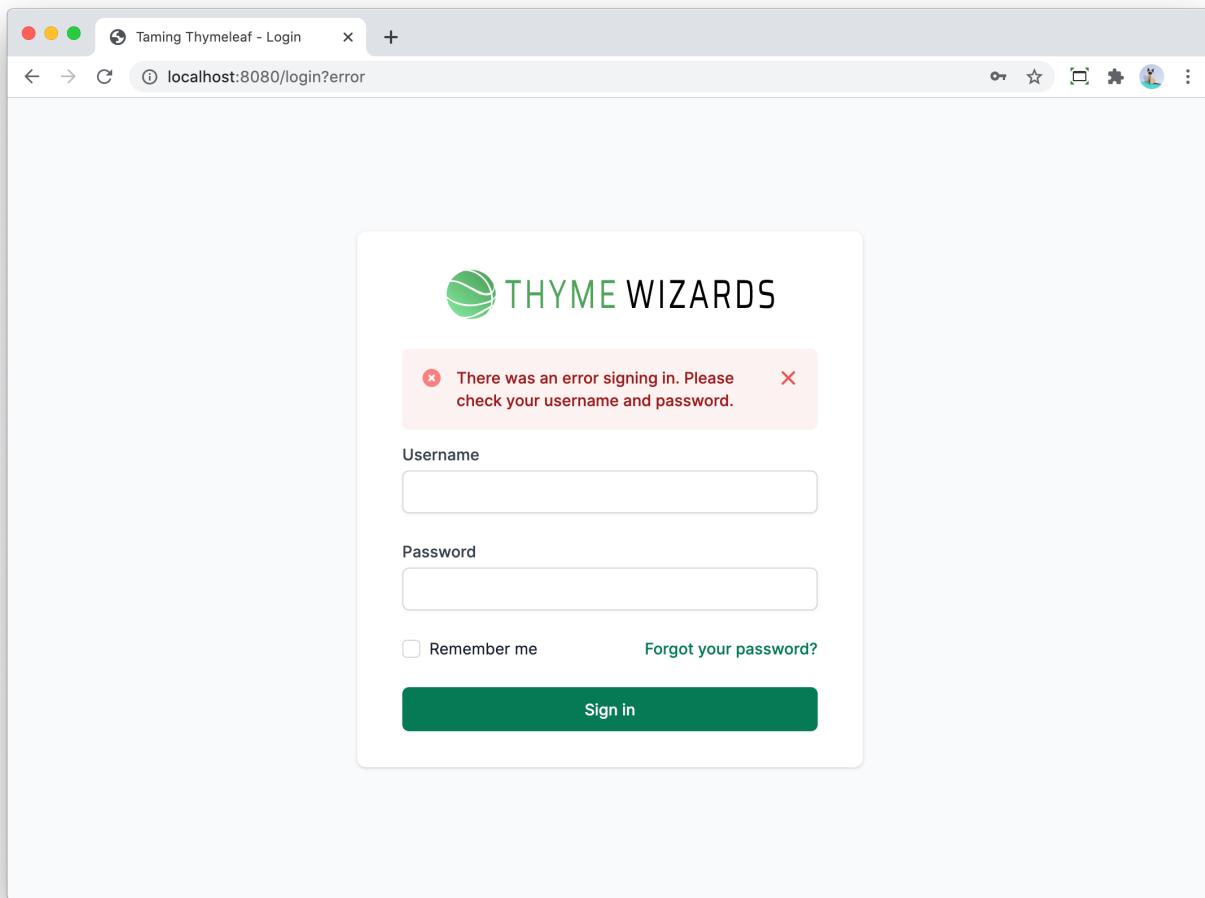


Figure 75. Error message when credentials are incorrect

## 14.6. Users from database

So far, our *security users* have been different from our *application users*. In this next section, we are going to connect our application users to Spring Security to combine them.

### 14.6.1. User entity updates

We will start with updating the `User` entity as we need to store 2 things extra for each `User` now:

- The password for the user
- The roles that are assigned to the user

We will represent the roles using an enum:

```
package com.tamingthymeleaf.application.user;
```

```
public enum UserRole {
    USER,
    ADMIN
}
```

Then we update the `User` entity by adding `roles` and `password` fields:

`com.tamingthymeleaf.application.user.User`

```
@Entity
@Table(name = "tt_user")
public class User extends AbstractVersionedEntity<UserId> {

    @ElementCollection(targetClass = UserRole.class) ①
    @Enumerated(EnumType.STRING) ②
    @CollectionTable(name = "user_roles") ③
    @Column(name = "role") ④
    private Set<UserRole> roles;

    @NotNull
    private String password; ⑤

    ...
}
```

- ① Instruct Hibernate that we want to store this as a collection of the "enumerated" basic type.
- ② We want to use the String representation of the enum (default is the ordinal of the enum).
- ③ Specify `user_roles` as the name of the collection database table.
- ④ Specify the name of the column where the enum value will be stored in the `user_roles` table.

Based on these changes, we now need to change our Flyway migration script:

`src/main/resources/db/migration/V1.0_init.sql`

```
CREATE TABLE tt_user
(
    id          UUID      NOT NULL,
    version     BIGINT    NOT NULL,
    password    VARCHAR   NOT NULL,
    first_name  VARCHAR   NOT NULL,
    last_name   VARCHAR   NOT NULL,
    gender      VARCHAR   NOT NULL,
    birthday    DATE      NOT NULL,
    email       VARCHAR   NOT NULL,
    phone_number VARCHAR  NOT NULL,
    PRIMARY KEY (id)
```

```
);

ALTER TABLE tt_user
    ADD CONSTRAINT UK_user_email UNIQUE (email);

CREATE TABLE user_roles
(
    user_id UUID      NOT NULL,
    role    VARCHAR NOT NULL
);

ALTER TABLE user_roles
    ADD CONSTRAINT FK_user_roles_to_user FOREIGN KEY (user_id)
REFERENCES tt_user;
```

 Because we *change* the migration script, we will need to clear the database. If you don't want that, create a new file [V1.1\\_\\_update-user-for-security.sql](#) which has the appropriate `ALTER TABLE` statements.

As we are still in early development phase, I usually edit the single Flyway script. Once we do a first release, this should of course no longer happen.

Still in `User.java`, we update the constructor with the extra fields and we create 2 factory methods: one to create a normal user, and one to create an administrator user:

`com.tamingthymeleaf.application.user.User`

```
private User(UserId id,
            Set<UserRole> roles,
            UserName userName,
            String password,
            Gender gender,
            LocalDate birthday,
            Email email,
            PhoneNumber phoneNumber) {
    super(id);
    this.roles = roles;
    this.userName = userName;
    this.password = password;
    this.gender = gender;
    this.birthday = birthday;
    this.email = email;
    this.phoneNumber = phoneNumber;
}

public static User createUser(UserId id,
```

```

        UserName userName,
        String encodedPassword,
        Gender gender,
        LocalDate birthday,
        Email email,
        PhoneNumber phoneNumber) {
    return new User(id, Set.of(UserRole.USER), userName,
encodedPassword, gender, birthday, email, phoneNumber);
}

public static User createAdministrator(UserId id,
                                      UserName userName,
                                      String encodedPassword,
                                      Gender gender,
                                      LocalDate birthday,
                                      Email email,
                                      PhoneNumber phoneNumber) {
    return new User(id, Set.of(UserRole.USER, UserRole.ADMIN),
userName, encodedPassword, gender, birthday, email, phoneNumber);
}

```

We also update `UserService` and `UserServiceImpl` to create users with the different roles:

`com.tamingthymeleaf.application.user.UserService`

```

User createUser(CreateUserParameters parameters);

User createAdministrator(CreateUserParameters parameters);

```

`com.tamingthymeleaf.application.user.UserService`

```

@Override
public User createUser(CreateUserParameters parameters) {
    LOGGER.debug("Creating user {} ({})", parameters.getUserName()
().getFullName(), parameters.getEmail().asString());
    UserId userId = repository.nextId();
    String encodedPassword = passwordEncoder.encode(parameters
.getPassword()); ①
    User user = User.createUser(userId,
                                parameters.getUserName(),
                                encodedPassword,
                                parameters.getGender(),
                                parameters.getBirthday(),
                                parameters.getEmail(),
                                parameters.getPhoneNumber());

```

```

        return repository.save(user);
    }

    @Override
    public User createAdministrator(CreateUserParameters parameters) {
        LOGGER.debug("Creating administrator {} {}", parameters
.getUserName().getFullName(), parameters.getEmail().asString());
        UserId userId = repository.nextId();
        User user = User.createAdministrator(userId,
                                            parameters.getUserName(),
                                            passwordEncoder.encode
(parameters.getPassword()),
                                            parameters.getGender(),
                                            parameters.getBirthday(),
                                            parameters.getEmail(),
                                            parameters.
getPhoneNumber());
        return repository.save(user);
    }
}

```

- ① The `CreateUserParameters` contain the password in clear text. We need to encode it using the `passwordEncoder` before we pass it to the `User.createUser()` factory method as we need the password to be encoded in the database.

Further, `com.tamingthymeleaf.application.user.CreateUserParameters` and `com.tamingthymeleaf.application.user.EditUserParameters` are also updated with the extra `password` field.

With this in place, we can update `DatabaseInitializer` to create some default users and an administrator:

`com.tamingthymeleaf.application.DatabaseInitializer`

```

@Component
@Profile("init-db")
public class DatabaseInitializer implements CommandLineRunner {
    private final Faker faker = new Faker();
    private final UserService userService;

    public DatabaseInitializer(UserService userService) {
        this.userService = userService;
    }

    @Override
    public void run(String... args) {
        for (int i = 0; i < 20; i++) { ①
            CreateUserParameters parameters = newRandomUserParameters();

```

```

        userService.createUser(parameters);
    }

    UserName userName = randomUserName();
    CreateUserParameters parameters = new CreateUserParameters
(userName,
userName.getFirstName(), ②
randomGender(),
LocalDate.parse("2000-01-01"),
generateEmailForUserName(userName),
randomPhoneNumber());
    userService.createAdministrator(parameters); ③
}

```

① Create 20 users

② Use the first name as password (This is obviously a bad security practise, but is ok since we only this for local testing)

③ Create 1 administrator

## 14.6.2. Spring Security connection

The next step is connecting our [User](#) entity with Spring Security.

We need 2 things to make that possible:

- An implementation of the [org.springframework.security.core.userdetails.UserDetailsService](#) interface.
- An implementation of the [org.springframework.security.core.userdetails.UserDetails](#) interface.

The [UserDetailsService](#) interface has a single method that takes a String. This parameter is the username or the email depending on what users should use to log on. The method returns an instance of [UserDetails](#) or throws an [org.springframework.security.core.userdetails.UsernameNotFoundException](#) if no user with the given username (or email) could be found:

[org.springframework.security.core.userdetails.UserDetailsService](#)

```

public interface UserDetailsService {
    UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException;
}

```

Create the [DatabaseUserDetailsService](#):

```
package com.tamingthymeleaf.application.infrastructure.security;

import com.tamingthymeleaf.application.user.Email;
import com.tamingthymeleaf.application.user.User;
import com.tamingthymeleaf.application.user.UserRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import
org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import static java.lang.String.format;

@Service
@Transactional(readOnly = true)
public class DatabaseUserDetailsService implements UserDetailsService {

    private final UserRepository userRepository;

    @Autowired
    public DatabaseUserDetailsService(UserRepository userRepository) {
①        this.userRepository = userRepository;
    }

    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        User user = userRepository.findByEmail(new Email(username)) ②
                .orElseThrow(() -> new
UsernameNotFoundException(
                                format("User with email %s
could not be found", username))); ③

        return new ApplicationUserDetails(user); ④
    }
}
```

① Inject the `UserRepository` to get the users from the database.

② Search for a user by email address as we will use emails for login.

- ③ Throw a `UsernameNotFoundException` if the returned `Optional` is empty.
- ④ Return an instance of  `ApplicationUserDetails` containing the information of the user from the database.

The `DatabaseUserDetailsService` returns an  `ApplicationUserDetails` instance which is defined as:

```
package com.tamingthymeleaf.application.infrastructure.security;

import com.tamingthymeleaf.application.user.User;
import com.tamingthymeleaf.application.user.UserId;
import org.springframework.security.core.GrantedAuthority;
import
org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;

import java.util.Collection;
import java.util.Set;
import java.util.stream.Collectors;

public class ApplicationUserDetails implements UserDetails { ①
    private final UserId id;
    private final String username;
    private final String displayName;
    private final String password;
    private final Set<GrantedAuthority> authorities;

    public ApplicationUserDetails(User user) {
        this.id = user.getId(); ②
        this.username = user.getEmail().asString(); ③
        this.displayName = user.getUserName().getFullName(); ④
        this.password = user.getPassword(); ⑤
        this.authorities = user.getRoles().stream()
            .map(userRole -> new
SimpleGrantedAuthority("ROLE_" + userRole.name()))
            .collect(Collectors.toSet()); ⑥
    }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return authorities;
    }

    @Override
    public String getPassword() {
```

```

        return password;
    }

    @Override
    public String getUsername() {
        return username;
    }

    @Override
    public boolean isAccountNonExpired() {
        return true;
    }

    @Override
    public boolean isAccountNonLocked() {
        return true;
    }

    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }

    @Override
    public boolean isEnabled() {
        return true;
    }

    public UserId getId() {
        return id;
    }

    public String getDisplayName() {
        return displayName;
    }
}

```

- ① Implement the `UserDetails` interface
- ② The `id` is not needed to satisfy the `UserDetails` interface, but it is usually convenient to have it.
- ③ We use the email address as `username`.
- ④ To have something to display nicely in the user interface, we keep track of the full name as `displayName` property.
- ⑤ The `password` is needed as Spring Security will use this to validate the password if the user logs on. Note that this is the encrypted password.

- ⑥ We need to map our `UserRole` enum to something that Spring Security understands, which are `GrantedAuthority` instances.

The last part of the puzzle is updating the `WebSecurityConfiguration` to make use of all this:

```
@Configuration
@EnableGlobalMethodSecurity(securedEnabled = true)
public class WebSecurityConfiguration extends
WebSecurityConfigurerAdapter {
    private final PasswordEncoder passwordEncoder;
    private final UserDetailsService userDetailsService;

    public WebSecurityConfiguration(PasswordEncoder passwordEncoder,
                                    UserDetailsService userDetailsService) { ①
        this.passwordEncoder = passwordEncoder;
        this.userDetailsService = userDetailsService;
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws
Exception {
    auth.userDetailsService(userDetailsService) ②
        .passwordEncoder(passwordEncoder); ③
}
...
}
```

① Inject the `UserDetailsService`

② Configure the `AuthenticationManagerBuilder` to use our `UserDetailsService`

③ Also pass in the `PasswordEncoder` so Spring Security can work with the encoded passwords in our database.

We can now test this.

Update `application.properties` with:

```
logging.level.com.tamingthymeleaf.application=DEBUG
```

So we can see the users that get created in the logging output.

It should look similar to this when running the application (Remember to use the `init-db` profile so users get created):

```
2020-08-29 09:13:41.455 INFO 43847 --- [           main]
c.t.a.TamingThymeleafApplication : Started
TamingThymeleafApplication in 3.473 seconds (JVM running for 3.938)
2020-08-29 09:13:41.566 DEBUG 43847 --- [           main]
c.t.application.user.UserServiceImpl : Creating user Charles Bartell
(charles.bartell@hotmail.com)
2020-08-29 09:13:41.720 DEBUG 43847 --- [           main]
c.t.application.user.UserServiceImpl : Creating user Evelyn Ortiz
(evelyn.ortiz@yahoo.com)
2020-08-29 09:13:41.811 DEBUG 43847 --- [           main]
c.t.application.user.UserServiceImpl : Creating user Joe Hahn
(joe.hahn@gmail.com)
2020-08-29 09:13:41.893 DEBUG 43847 --- [           main]
c.t.application.user.UserServiceImpl : Creating user Patrick Nolan
(patrick.nolan@yahoo.com)
2020-08-29 09:13:41.979 DEBUG 43847 --- [           main]
c.t.application.user.UserServiceImpl : Creating user Larisa Gibson
(larisa.gibson@yahoo.com)
2020-08-29 09:13:42.059 DEBUG 43847 --- [           main]
c.t.application.user.UserServiceImpl : Creating user Robert Turcotte
(robert.turcotte@yahoo.com)
2020-08-29 09:13:42.139 DEBUG 43847 --- [           main]
c.t.application.user.UserServiceImpl : Creating user Honey Witting
(honey.witting@yahoo.com)
2020-08-29 09:13:42.219 DEBUG 43847 --- [           main]
c.t.application.user.UserServiceImpl : Creating user Julieann Hintz
(julieann.hintz@yahoo.com)
2020-08-29 09:13:42.303 DEBUG 43847 --- [           main]
c.t.application.user.UserServiceImpl : Creating user Idella Hammes
(idella.hammes@hotmail.com)
2020-08-29 09:13:42.384 DEBUG 43847 --- [           main]
c.t.application.user.UserServiceImpl : Creating user Sidney
Runolfsson (sidney.runolfsson@yahoo.com)
2020-08-29 09:13:42.463 DEBUG 43847 --- [           main]
c.t.application.user.UserServiceImpl : Creating user Randolph
Denesik (randolph.denesik@gmail.com)
2020-08-29 09:13:42.542 DEBUG 43847 --- [           main]
c.t.application.user.UserServiceImpl : Creating user Jonah Harber
(jonah.harber@yahoo.com)
2020-08-29 09:13:42.629 DEBUG 43847 --- [           main]
c.t.application.user.UserServiceImpl : Creating user Josue Herman
(josue.herman@yahoo.com)
2020-08-29 09:13:42.707 DEBUG 43847 --- [           main]
c.t.application.user.UserServiceImpl : Creating user Marlin Hilpert
```

```
(marlin.hilpert@gmail.com)
2020-08-29 09:13:42.787 DEBUG 43847 --- [           main]
c.t.application.user.UserServiceImpl : Creating user Hildegarde
Borer (hildegarde.borer@gmail.com)
2020-08-29 09:13:42.867 DEBUG 43847 --- [           main]
c.t.application.user.UserServiceImpl : Creating user Hollis O'Reilly
(hollis.oreilly@yahoo.com)
2020-08-29 09:13:42.948 DEBUG 43847 --- [           main]
c.t.application.user.UserServiceImpl : Creating user Bess Dietrich
(bess.dietrich@gmail.com)
2020-08-29 09:13:43.029 DEBUG 43847 --- [           main]
c.t.application.user.UserServiceImpl : Creating user Davis Jast
(davis.jast@hotmail.com)
2020-08-29 09:13:43.115 DEBUG 43847 --- [           main]
c.t.application.user.UserServiceImpl : Creating user Solomon Windler
(solomon.windler@yahoo.com)
2020-08-29 09:13:43.194 DEBUG 43847 --- [           main]
c.t.application.user.UserServiceImpl : Creating user Mathew
Gulgowski (mathew.gulgowski@yahoo.com)
2020-08-29 09:13:43.282 DEBUG 43847 --- [           main]
c.t.application.user.UserServiceImpl : Creating administrator
Shamika Olson (shamika.olson@hotmail.com)
```

You should now be able to log on using any of those users. For example, use [davis.jast@hotmail.com](#) with password `Davis` to log on as a user. Or use [shamika.olson@hotmail.com](#) with password `Shamika` to log on as an administrator.

#### 14.6.3. Show current user info

We have seen in the previous chapter that we could show the username on a Thymeleaf page using `sec:authentication="name"` and the roles of the user with `sec:authentication="principal.authorities"`

The `principal` refers to the `UserDetails` implementation that is returned by the `UserDetailsService`. We can call any method of our  `ApplicationUserDetails` implementation in the `sec:authentication` attribute to display information about the current user.

As an example, we can update `top-menu.html` to show the `displayName` and the `username` information from  `ApplicationUserDetails`:

```
<div class="block px-4 pt-1 text-sm text-gray-700"
    sec:authentication="principal.displayName"></div> ①
<div class="block px-4 pb-2 text-xs font-mono text-gray-400 border-b
truncate"
    sec:authentication="principal.username"></div> ②
```

- ① Show the `displayName` of the logged on user
- ② Show the `username` of the logged on user (Which is actually the email address in this case)

The screenshot shows a web application titled "Taming Thymeleaf - Users" running on "localhost:8080/users". The left sidebar contains navigation links: Dashboard, Users (selected), Teams, Calendar, Documents, and Reports. The main content area displays a table of users with columns: NAME, GENDER, BIRTHDAY, and EMAIL. A user named "Davis Jast" is selected, and a context menu is open over their row. The menu items are: Profile, Settings, and Sign out. The user's profile picture is also visible next to the menu.

NAME	GENDER	BIRTHDAY	EMAIL
Charles Bartell	FEMALE	1994-01-02	charles.bartel...
Hildegarde Borer	MALE	2000-04-19	hildegard...
Randolph Denesik	FEMALE	2003-07-19	randolph.denesik@gmail.com
Bess Dietrich	FEMALE	1998-03-15	bess.dietrich@gmail.com
Larisa Gibson	FEMALE	1991-12-31	larisa.gibson@yahoo.com
Mathew Gulgowski	FEMALE	2009-10-15	mathew.gulgowski@yahoo.com
Joe Hahn	MALE	1998-01-04	joe.hahn@gmail.com
Idella Hammes	MALE	1986-12-14	idella.hammes@hotmail.com
Jonah Harber	MALE	1985-09-12	jonah.harber@yahoo.com
Josue Herman	FEMALE	1986-01-23	josue.herman@yahoo.com

Showing 1 to 10 of 21 results

Figure 76. Popup menu showing the display name and the email address of the logged on user

#### 14.6.4. Create user form

We used the `DatabaseInitializer` to create some users, but now we also need to make the 'create user' form work again with the 2 roles and the password field.

We'll start with adding the user role selection. We will use a HTML `<select>` for this (also sometimes called a dropdown or combobox).

Add the following to `edit.html`:

`src/main/resources/templates/users/edit.html`

```

<div class="sm:col-span-2">
    <label for="userRole" class="block text-sm font-medium text-gray-700">User
        Role</label>
    <select id="userRole"
        class="max-w-lg block focus:ring-green-500 focus:border-green-500 w-full shadow-sm sm:max-w-xs sm:text-sm border-gray-300"

```

```

rounded-md"
    th:field="*{userRole}"> ①
    <option th:each="role : ${possibleRoles}"
        th:text="#{'UserRole.' + ${role.name()}}"
        th:value="${role.name()}>User ②
    </option>
</select>
</div>

```

① The `<select>` needs to bind to the `userRole` field in the `CreateUserData` object.

② We need to add an `<option>` tag for each possible role. We will update the `UserController` to add the list of possible roles in the model under the `possibleRoles` key. This list will contain `UserRole` enum instances.

The `value` attribute of the `<option>` is the `name()` of the `UserRole` enum. The `value` should always be something fixed that is not translated. So the `name()` of an enum is a good candidate, or a primary key value for example.

For the text that we show to the user, we use the `#{'UserRole.' + ${role.name()}}` construction. This allows to translate the enum values like this in `messages.properties`:

```

UserRole.USER=User
UserRole.ADMIN=Administrator

```

Next, we add 2 password fields to `edit.html`

```

<div th:replace="fragments/forms :: textinput(#{user.firstName},
'firstName', 'sm:col-span-3')"></div>
<div th:replace="fragments/forms :: textinput(#{user.lastName},
'lastName', 'sm:col-span-3')"></div>
<div th:replace="fragments/forms :: textinput(labelText=#{user.email},
fieldName='email', cssClass='sm:col-span-4', inputType='email')"></div>
<div th:replace="fragments/forms :: textinput(labelText=#{user.password}, fieldName='password',
cssClass='sm:col-span-4', inputType='password')"></div> ①
<div th:replace="fragments/forms :: textinput(labelText=#{user.password.repeated},
fieldName='passwordRepeated', cssClass='sm:col-span-4',
inputType='password')"></div> ②
<div th:replace="fragments/forms :: textinput(#{user.phoneNumber},
'phoneNumber', 'sm:col-span-4')"></div>
<div class="sm:col-span-2"></div>
<div th:replace="fragments/forms :: textinput(labelText=#{user.birthday}, fieldName='birthday',
cssClass='sm:col-span-2',

```

```
placeholder="#{user.birthday.placeholder})"></div>
```

- ① We can re-use the `textinput` fragment with an `inputType` of `password` to easily add the password field to the form.
- ② To ensure the administrator has no typo while typing the password for the user, they will be required to enter the password twice.

To support the form, we need to update `CreateUserData`:

```
@NotExistingUser(groups = ValidationGroupTwo.class)
@PasswordsMatch(groups = ValidationGroupTwo.class)
public class CreateUserData {
    @NotNull
    private UserRole userRole; ①
    @NotBlank
    @Size(min = 1, max = 200, groups = ValidationGroupOne.class)
    private String firstName;
    @NotBlank
    @Size(min = 1, max = 200, groups = ValidationGroupOne.class)
    private String lastName;
    @NotBlank
    private String password; ②
    @NotBlank
    private String passwordRepeated; ③
    ...
}
```

- ① The `userRole` field for the `<select>`
- ② The `password` field for the password `<input>`
- ③ The `passwordRepeated` field to capture the repeated password `<input>`

Next to the new fields in the `CreateUserData` class, we see an extra validation annotation `@PasswordsMatch` that we will use to validate if `password` and `passwordRepeated` are exactly the same:

```
package com.tamingthymeleaf.application.user.web;

import javax.validation.Constraint;
import javax.validation.Payload;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(ElementType.TYPE)
```

```

@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = PasswordsMatchValidator.class)
public @interface PasswordsMatch {
    String message() default "{PasswordsNotMatching}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};
}

```

The validator that handles the validation logic is [PasswordsMatchValidator](#):

```

package com.tamingthymeleaf.application.user.web;

import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;

public class PasswordsMatchValidator implements ConstraintValidator<PasswordsMatch, CreateUserData> {
    @Override
    public void initialize(PasswordsMatch constraintAnnotation) {

    }

    @Override
    public boolean isValid(CreateUserData value,
    ConstraintValidatorContext context) {
        if (!value.getPassword().equals(value.getPasswordRepeated())) {
            context.disableDefaultConstraintViolation();
            context.buildConstraintViolationWithTemplate
                    ("{PasswordsNotMatching}")
                        .addPropertyNode("passwordRepeated")
                        .addConstraintViolation();

            return false;
        }

        return true;
    }
}

```

By using `{PasswordsNotMatching}` in the `buildConstraintViolationWithTemplate`, we need to add the actual error message in `messages.properties`:

**PasswordsNotMatching=The passwords do not match. Please ensure you type the same password twice.**

Like that, this can be translated if needed.

Finally, we need to update `UserController` to add the `possibleRoles` to the model. We need to do this in the `@GetMapping` and `@PostMapping` for the both the create and the edit situation.

This is an example for the `@GetMapping` of `/users/create`, but the others are similar:

`com.tamingthymeleaf.application.user.web.UserController`

```

@GetMapping("/create")
@Secured("ROLE_ADMIN")
public String createUserForm(Model model) {
    model.addAttribute("user", new CreateUserFormData());
    model.addAttribute("genders", List.of(Gender.MALE, Gender.FEMALE, Gender.OTHER));
    model.addAttribute("possibleRoles", List.of(UserRole.values()));
①
    model.addAttribute("editMode",EditMode.CREATE);
    return "users/edit";
}

```

① Add all `UserRole` enum values under the `possibleRoles` key.

The create user form will now render like this with our extra fields:

The screenshot shows a web application interface for creating a new user. On the left, there's a sidebar with a logo and several navigation items: Dashboard, Users (which is currently selected), Teams, Calendar, Documents, and Reports. The main content area has a title "Create user". It includes a dropdown menu for "User Role" set to "User", three radio buttons for "Gender" (Male, Female, Other), and input fields for "First name", "Last name", "Email", "Password", "Password (Repeated)", and "Phone number". A small profile picture of a person is visible in the top right corner of the main window.

Figure 77. Create user form with role selection and password fields

When the passwords do not match, we get an appropriate error message:

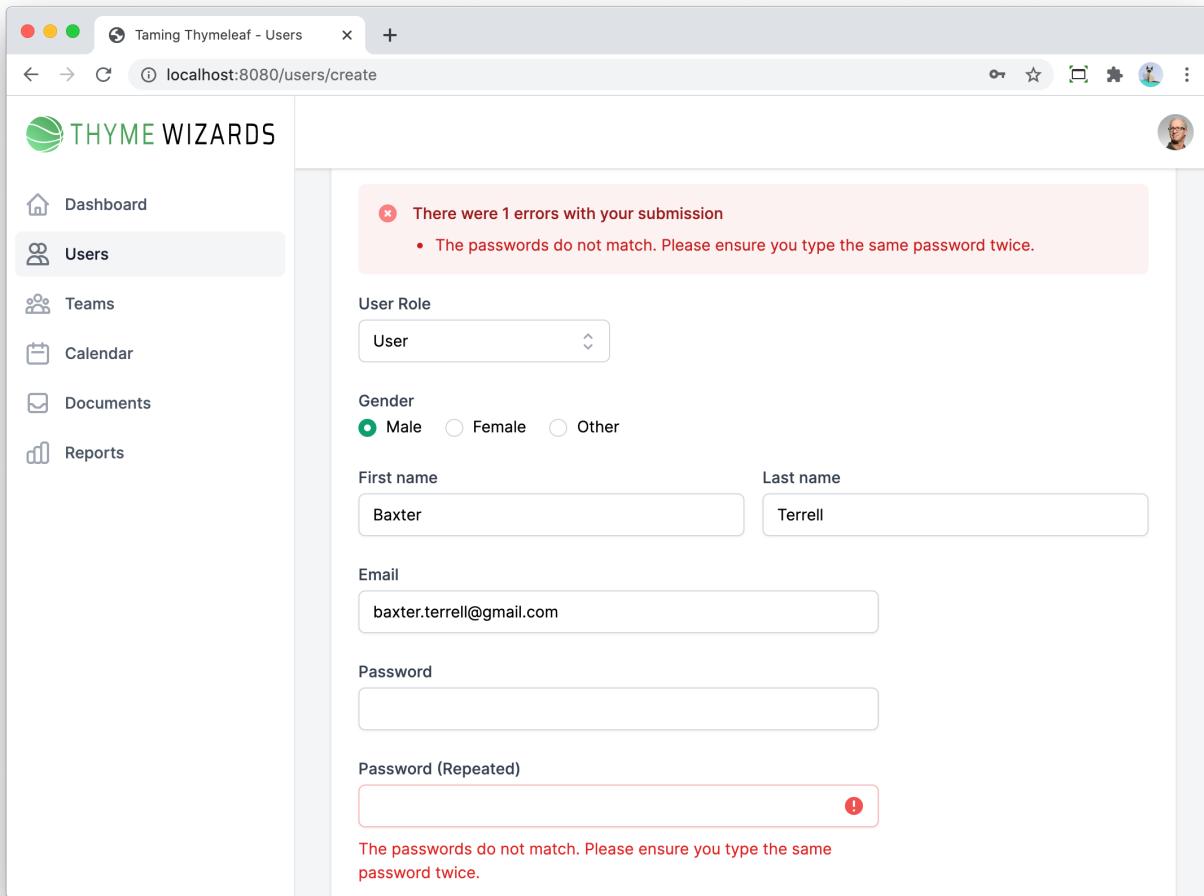


Figure 78. Error message when passwords do not match

This works now fine for creating new users, but due to `edit.html` being used for creating *and* editing users and the `EditUserData` extending `CreateUserData`, we now have to enter a password just for editing a user as well. Probably not what we want.

#### 14.6.5. Refactor the edit user implementation

To avoid having to enter the password of the user when editing user details, we will do a small refactor.

We start by creating an abstract super class from `CreateUserData` extracting most fields, except for `password` and `passwordRepeated`:

```
package com.tamingthymeleaf.application.user.web;

import
com.tamingthymeleaf.application.infrastructure.validation.ValidationGrou
pOne;
import
com.tamingthymeleaf.application.infrastructure.validation.ValidationGrou
pTwo;
import com.tamingthymeleaf.application.user.Gender;
```

```
import com.tamingthymeleaf.application.user.UserRole;
import org.springframework.format.annotation.DateTimeFormat;

import javax.validation.constraints.*;
import java.time.LocalDate;

@NotExistingUser(groups = ValidationGroupTwo.class)
public class AbstractUserData {
    @NotNull
    private UserRole userRole; ①
    @NotBlank
    @Size(min = 1, max = 200, groups = ValidationGroupOne.class)
    private String firstName;
    @NotBlank
    @Size(min = 1, max = 200, groups = ValidationGroupOne.class)
    private String lastName;
    @NotNull
    private Gender gender;
    @NotBlank
    @Email(groups = ValidationGroupOne.class)
    private String email;
    @NotNull
    @DateTimeFormat(pattern = "yyyy-MM-dd")
    private LocalDate birthday;
    @NotBlank
    @Pattern(regexp = "[0-9.\\-() x/+]+", groups = ValidationGroupOne
.class)
    private String phoneNumber;

    public UserRole getUserRole() {
        return userRole;
    }

    public void setUserRole(UserRole userRole) {
        this.userRole = userRole;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
}
```

```

public String getLastName() {
    return lastName;
}

public void setLastName(String lastName) {
    this.lastName = lastName;
}

public Gender getGender() {
    return gender;
}

public void setGender(Gender gender) {
    this.gender = gender;
}

public String getEmail() {
    return email;
}

public void setEmail(String email) {
    this.email = email;
}

public LocalDate getBirthday() {
    return birthday;
}

public void setBirthday(LocalDate birthday) {
    this.birthday = birthday;
}

public String getPhoneNumber() {
    return phoneNumber;
}

public void setPhoneNumber(String phoneNumber) {
    this.phoneNumber = phoneNumber;
}
}

```

We keep `password` and `passwordRepeated` fields on `CreateUserData`:

```
package com.tamingthymeleaf.application.user.web;
```

```
import
com.tamingthymeleaf.application.infrastructure.validation.ValidationGrou
pTwo;
import com.tamingthymeleaf.application.user.CreateUserParameters;
import com.tamingthymeleaf.application.user.PhoneNumber;
import com.tamingthymeleaf.application.user.UserName;

import javax.validation.constraints.NotBlank;

@PasswordsMatch(groups = ValidationGroupTwo.class) ①
public class CreateUserData extends AbstractUserFormData {
    @NotBlank
    private String password;
    @NotBlank
    private String passwordRepeated;

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getPasswordRepeated() {
        return passwordRepeated;
    }

    public void setPasswordRepeated(String passwordRepeated) {
        this.passwordRepeated = passwordRepeated;
    }

    public CreateUserParameters toParameters() {
        return new CreateUserParameters(new UserName(getFirstName(),
getLastName(),
                                password,
                                getGender(),
                                getBirthday(),
                                new com.tamingthymeleaf
.application.user.Email(getEmail()),
                                new PhoneNumber(
getPhoneNumber())));
    }
}
```

```
}
```

- ① The `@PasswordsMatch` validation annotation remains on `CreateUserFormData` since it is only relevant when creating a user account.

The `@NotExistingUser` validation annotation is moved to `AbstractUserFormData` since we need to check this when creating a user and when editing a user. Due to this, we also need to update the generics for the validator class:

```
package com.tamingthymeleaf.application.user.web;

import com.tamingthymeleaf.application.user.Email;
import com.tamingthymeleaf.application.user.UserService;
import org.springframework.beans.factory.annotation.Autowired;

import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;

public class NotExistingUserValidator implements ConstraintValidator<NotExistingUser, AbstractUserFormData> { ①

    private final UserService userService;

    @Autowired
    public NotExistingUserValidator(UserService userService) {
        this.userService = userService;
    }

    public void initialize(NotExistingUser constraint) {
        // intentionally empty
    }

    public boolean isValid(AbstractUserFormData formData,
                          ConstraintValidatorContext context) { ②
        if (userService.userWithEmailExists(new Email(formData.
getEmail()))) {
            context.disableDefaultConstraintViolation();
            context.buildConstraintViolationWithTemplate
                    ("{UserAlreadyExisting}")
                    .addPropertyNode("email")
                    .addConstraintViolation();
        }
        return false;
    }

    return true;
}
```

```
}
```

① Use `AbstractUserData` in the generics.

② `isValid` needs to match with the updated generics.

We have `EditUserData` now extend from `AbstractUserData` (instead of `CreateUserData`):

```
package com.tamingthymeleaf.application.user.web;

import com.tamingthymeleaf.application.user.*;

public class EditUserData extends AbstractUserData { ①
    private String id;
    private long version;

    public static EditUserData fromUser(User user) {
        EditUserData result = new EditUserData();
        result.setId(user.getId().asString());
        result.setVersion(user.getVersion());
        result.setFirstName(user.getUserName().getFirstName());
        result.setLastName(user.getUserName().getLastName());
        result.setGender(user.getGender());
        result.setBirthday(user.getBirthday());
        result.setEmail(user.getEmail().asString());
        result.setPhoneNumber(user.getPhoneNumber().asString());

        return result;
    }

    public EditUserParameters toParameters() {
        return new EditUserParameters(version,
            new UserName(getFirstName(),
getLastName(),
                getGender(),
                getBirthday(),
                new Email(getEmail()),
                new PhoneNumber(
getPhoneNumber())));
    }

    public String getId() {
        return id;
    }
}
```

```

public void setId(String id) {
    this.id = id;
}

public long getVersion() {
    return version;
}

public void setVersion(long version) {
    this.version = version;
}
}

```

① Use `AbstractUserFormData` as base class

Finally, we update the `edit.html` to only have the password fields upon creation:

```

<div th:replace="fragments/forms :: textinput(#{user.firstName},
'firstName', 'sm:col-span-3')"></div>
<div th:replace="fragments/forms :: textinput(#{user.lastName},
'lastName', 'sm:col-span-3')"></div>
<div th:replace="fragments/forms :: textinput(labelText=#{user.email},
fieldName='email', cssClass='sm:col-span-4', inputType='email')"></div>
<th:block th:if="${editMode?.name() == 'CREATE'}"> ①
    <div th:replace="fragments/forms :: textinput(labelText=#{user.password},
    fieldName='password', cssClass='sm:col-span-4', inputType='password')"></div>
    <div th:replace="fragments/forms :: textinput(labelText=#{user.password.repeated},
    fieldName='passwordRepeated', cssClass='sm:col-span-4',
    inputType='password')"></div>
</th:block>
<div th:replace="fragments/forms :: textinput(#{user.phoneNumber},
'phoneNumber', 'sm:col-span-4')"></div>
<div class="sm:col-span-2"></div>
<div th:replace="fragments/forms :: textinput(labelText=#{user.
birthday}, fieldName='birthday', cssClass='sm:col-span-2', placeholder=
#{user.birthday.placeholder})"></div>

```

① Use the `editMode` model property to render the password fields when the template is used in `CREATE` mode.

Editing a user now no longer displays the password fields:

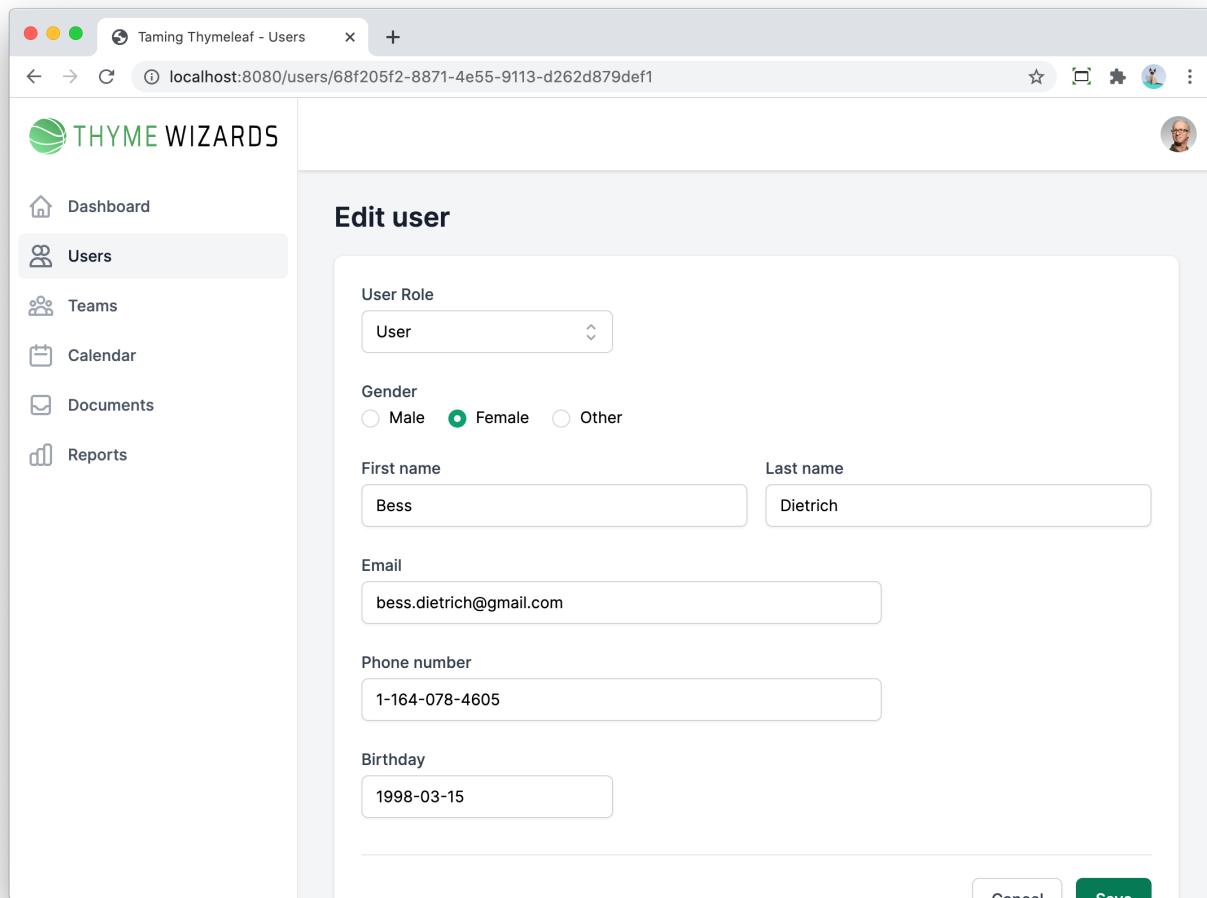


Figure 79. Edit user does not display the password fields

## 14.7. Summary

In this chapter, you learned:

- How to add Spring Security to a Spring Boot project.
- How to configure authentication in Spring Security.
- How to configure authorization in Spring Security.
- How to link Spring Security users with the application users.



We have only scratched the surface of all that is possible with Spring Security. I would recommend taking a look at [the reference documentation](#) or [one of the various books](#) on the topic for more information.

# Chapter 15. Testing

We haven't done much automated testing so far, except for the database test in [the chapter on Spring Data JPA](#). It is equally important that we cover our Thymeleaf templates and controllers with automated tests. We will see the various options that Spring offers for testing, and how and when to use them.

## 15.1. Using `@WebMvcTest`

Spring Boot offers various utilities to start an application for testing, either fully or partially. We can use `@SpringBootTest` as a class annotation for a JUnit test and this will start our complete application, including an embedded Tomcat to serve the HTML pages. However, for tests that will only test a controller for example, we don't need (or want) the full application started.

Spring Boot has the concept of [test slices](#) that allows to start for example only the database layer (`@DataJpaTest`), or only the web layer (`@WebMvcTest`), or only the JSON converters (`@JsonTest`), ....

Here we will use `@WebMvcTest` since this is the test slice want for testing our [UserController](#). The `@WebMvcTest` annotation will automatically configure a mock servlet environment. Using [MockMvc](#), we can interact with our controller and validate requests and responses.

The following things will be created using `@WebMvcTest`:

- All `@Controller` beans (or only a single one when specifying the class name on the `@WebMvcTest` annotation).
- All `@ControllerAdvice` beans.
- All `@JsonComponent` beans so custom JSON serializers and deserializers will be active.
- `org.springframework.core.convert.converter.Converter` beans.
- `Filter` beans
- `WebMvcConfigurer` beans
- `HandlerMethodArgumentResolver` beans
- Spring Security configuration beans

What is *not* created?

- `@Component`
- `@Service`
- `@Repository`
- `@Configuration`

If your controller needs any of those, you should:

- either add them by using `@Import`,
- add mock implementations by using `@MockBean`,
- or use `@TestConfiguration` with `@Bean`.

*Why are we not doing plain unit tests for controllers?*

You might wonder why we are immediately doing an integration test and not a plain unit test. In the case of a controller, there is quite a lot going on in terms of annotations that are interpreted by the Spring Framework and security configuration of the application.



Moreover, as a good rule of thumb, a controller should not contain much logic, but delegate to helper classes (Services and/or repositories).

Given all that, the value of a plain unit test would be very limited.

### 15.1.1. Getting started with `@WebMvcTest`

Let's start with an example `@WebMvcTest`:

```
package com.tamingthymeleaf.application.user.web;

import com.tamingthymeleaf.application.user.UserService;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.context.annotation.Bean;
import
org.springframework.security.crypto.factory.PasswordEncoderFactories;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.test.web.servlet.MockMvc;

import static org.springframework.test.web.servlet.request
.MockMvcBuilders.get;
import static org.springframework.test.web.servlet.result
.MockMvcResultHandlers.print;
import static org.springframework.test.web.servlet.result
.MockMvcResultMatchers.redirectedUrl;
import static org.springframework.test.web.servlet.result
.MockMvcResultMatchers.status;

@WebMvcTest(UserController.class)
①
class UserControllerTest {

    @Autowired
    private MockMvc mockMvc; ②
    @MockBean
```

```

private UserService userService; ③

@Test
void testGetUsersRedirectsToLoginWhenNotAuthenticated() throws
Exception { ④
    mockMvc.perform(get("/users")) ⑤
        .andDo(print()) ⑥
        .andExpect(status().is3xxRedirection()) ⑦
        .andExpect(redirectedUrl("http://localhost/login")); ⑧
}

@TestConfiguration
static class TestConfig { ⑨
    @Bean
    public PasswordEncoder passwordEncoder() { ⑩
        return PasswordEncoderFactories
.createDelegatingPasswordEncoder();
    }
}
}

```

- ① Annotate the test class with `@WebMvcTest` so the testing infrastructure will be started. We indicate what controller we want to test by adding the class name of our `UserController` as an argument to the annotation.



If we don't specify any controller class, then all controllers of the application will be loaded.

- ② Spring Test will automatically configure a `MockMvc` instance that we can autowire.
- ③ Our `UserController` has a dependency of a `UserService`. Spring Test will *not* create the `UserServiceImpl` bean automatically, so either we need to create it ourselves, or we can ask Mockito to create a mock instance. Usually, you will create a mock instance and add it to the test context. This is easily done by using the `@MockBean` annotation.
- ④ Regular JUnit 5 test method.
- ⑤ Using `mockMvc.perform()`, we can issue `GET`, `POST`, ... requests at a certain URL. We used a static import of `org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get` to make the code fluent to read.
- ⑥ Using `andDo(print())`, we can print the request and the response. This can be convenient for debugging purposes. The `print()` method is statically imported from `org.springframework.test.web.servlet.result.MockMvcResultHandlers.print`.
- ⑦ We can add expectations about the response status via the `andExpect()` method on `MockMvc` in combination with the static `status()` method. The `status` method is statically imported from `org.springframework.test.web.servlet.result.MockMvcResultMatchers.status`.
- ⑧ Create an inner class annotated with `@TestConfiguration`. The Spring testing framework will pick up this class and augment the Spring context that is created due to the `@WebMvcTest`

annotation.

- ⑨ Create a [PasswordEncoder](#) bean, which will be added to the test context. We need this since the test will import our [WebSecurityConfiguration](#) as the testing framework will not only create the controller, but also the web security related beans. In the production application, we create this in [TamingThymeleafApplicationConfiguration](#), but in the test, this class is not loaded, so we need to add it here.

If we now run this test, it should be ok and will output something like:

```

MockHttpServletRequest:
    HTTP Method = GET
    Request URI = /users
    Parameters = {}
    Headers = []
        Body = <no character encoding set>
    Session Attrs =
{SPRING_SECURITY_SAVED_REQUEST=DefaultSavedRequest[http://localhost/users]}

Handler:
    Type = null

Async:
    Async started = false
    Async result = null

Resolved Exception:
    Type = null

ModelAndView:
    View name = null
        View = null
        Model = null

FlashMap:
    Attributes = null

MockHttpServletResponse:
    Status = 302
    Error message = null
    Headers = [X-Content-Type-Options:"nosniff", X-XSS-
Protection:"1; mode=block", Cache-Control:"no-cache, no-store, max-
age=0, must-revalidate", Pragma:"no-cache", Expires:"0", X-Frame-
Options:"DENY", Location:"http://localhost/login"]
    Content type = null
    Body =

```

```
Forwarded URL = null
Redirected URL = http://localhost/login
Cookies = []
```

Because we have our security configured and we are not authenticated in our test, we expect to be redirected to the login page which is exactly what we test here.

The printed details of the request and response show the following information:

- **MockHttpServletRequest**: Details about the request that was done including any parameters or headers that are sent.
- **Handler**: This normally shows the class that handled the request. Because Spring Security handled the request before it got to our handlers in this test, nothing is shown here.
- **Async**
- **Resolved Exception**: The exception class when there was an exception during the request processing.
- **ModelAndView**: Details about the model parameters and the view template.
- **FlashMap**: Details about the **flash attributes** if they are used.
- **MockHttpServletResponse**: Details about the response that was returned, including headers and the response body.

### 15.1.2. Authenticating in the `@WebMvcTest`

Since our **UserController** has all its methods secured, we need a way to simulate a user logging in so we can do some actual validation on the logic of the controller.

Spring Security test has some helper annotations like `@WithMockUser` and `@WithUserDetails` to help us. In our production security configuration, we use `DatabaseUserDetailsService` and `ApplicationUserDetails` classes. As we don't want to involve a database, we can create a `StubUserDetailsService` alternative for the automated test:

```
package com.tamingthymeleaf.application.infrastructure.security;

import com.tamingthymeleaf.application.user.*;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import
org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.security.crypto.password.PasswordEncoder;

import java.time.LocalDate;
import java.util.HashMap;
import java.util.Map;
import java.util.Optional;
import java.util.UUID;
```

```

public class StubUserDetailsService implements UserDetailsService { ①
    public static final String USERNAME_USER = "alanna.sparrow@hey.com";
    public static final String USERNAME_ADMIN = "gavin.joyce@gmail.com";

    private final Map<String, ApplicationUserDetails> users = new
HashMap<>(); ②

    public StubUserDetailsService(PasswordEncoder passwordEncoder) { ③
        addUser(new ApplicationUserDetails(createUser(
passwordEncoder)));
        addUser(new ApplicationUserDetails(createAdmin(
passwordEncoder)));
    }

    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        return Optional.ofNullable(users.get(username)) ④
            .orElseThrow(() -> new UsernameNotFoundException
(username));
    }

    private void addUser(ApplicationUserDetails userDetailsService) {
        users.put(userDetailsService.getUsername(), userDetailsService);
    }

    private User createUser(PasswordEncoder passwordEncoder) {
        return User.createUser(new UserId(UUID.randomUUID()),
            new UserName("Alanna", "Sparrow"),
            passwordEncoder.encode("secret"),
            Gender.FEMALE,
            LocalDate.parse("2001-06-19"),
            new Email(USERNAME_USER),
            new PhoneNumber("+555 123 456"));
    }

    private User createAdmin(PasswordEncoder passwordEncoder) {
        return User.createAdministrator(new UserId(UUID.randomUUID()),
            new UserName("Gavin", "Joyce"),
            passwordEncoder.encode(
"secret"),
            Gender.MALE,
            LocalDate.parse("2001-06-19"),
            new Email(USERNAME_ADMIN),
            new PhoneNumber("+555 123

```

```
456"));
}
}
```

- ① Implement `UserDetailsService` like our production `DatabaseUserDetailsService`.
- ② Keep the users in a `Map` in memory.
- ③ Add 2 users in the constructor so they are available for the tests.
- ④ Use the `users` map to search for a matching user.

We create a 'user' user and an 'admin' user, so we can test both security roles. Using that `StubUserDetailsService` helper class, we can write a test that uses it:

```
package com.tamingthymeleaf.application.user.web;

import
com.tamingthymeleaf.application.infrastructure.security.StubUserDetailsService;
import com.tamingthymeleaf.application.user.UserService;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.context.annotation.Bean;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.security.core.userdetails.UserDetailsService;
import
org.springframework.security.crypto.factory.PasswordEncoderFactories;
import org.springframework.security.crypto.password.PasswordEncoder;
import
org.springframework.security.test.context.support.WithUserDetails;
import org.springframework.test.web.servlet.MockMvc;
import
org.thymeleaf.spring5.templateresolver.SpringResourceTemplateResolver;
import org.thymeleaf.templateresolver.ITemplateResolver;

import static com.tamingthymeleaf.application.infrastructure.security
.StubUserDetailsService.USERNAME_USER;
import static org.mockito.ArgumentMatchers.any;
import static org.mockito.Mockito.when;
import static org.springframework.test.web.servlet.request
.MockMvcBuilders.get;
```

```

import static org.springframework.test.web.servlet.result
.MockMvcResultHandlers.print;
import static org.springframework.test.web.servlet.result
.MockMvcResultMatchers.redirectedUrl;
import static org.springframework.test.web.servlet.result
.MockMvcResultMatchers.status;

@WebMvcTest(UserController.class)
class UserControllerTest {

    @Autowired
    private MockMvc mockMvc;
    @MockBean
    private UserService userService;

    @Test
    void testGetUsersRedirectsToLoginWhenNotAuthenticated() throws
Exception {
        mockMvc.perform(get("/users"))
            .andDo(print())
            .andExpect(status().is3xxRedirection())
            .andExpect(redirectedUrl("http://localhost/login"));
    }

    @Test
    @WithUserDetails(USERNAME_USER) ①
    void testGetUsersAsUser() throws Exception {

        when(userService.getUsers(any(Pageable.class))) ②
            .thenReturn(Page.empty());

        mockMvc.perform(get("/users"))
            .andDo(print())
            .andExpect(status().isOk()); ③
    }

    @TestConfiguration
    static class TestConfig {
        @Bean
        public PasswordEncoder passwordEncoder() {
            return PasswordEncoderFactories
.createDelegatingPasswordEncoder();
        }
    }

    @Bean
}

```

```

    public ITemplateResolver svgTemplateResolver() { ④
        SpringResourceTemplateResolver resolver = new
        SpringResourceTemplateResolver();
        resolver.setPrefix("classpath:/templates/svg/");
        resolver.setSuffix(".svg");
        resolver.setTemplateMode("XML");

        return resolver;
    }

    @Bean
    public UserDetailsService userDetailsService(PasswordEncoder
passwordEncoder) { ⑤
        return new StubUserDetailsService(passwordEncoder);
    }
}

}

```

- ① Use `@WithUserDetails` to instruct Spring Security test to simulate that there is a logged on user. The passed in parameter of the annotation will be used to ask the `UserDetailsService` for the user.
- ② Setup Mockito so that the call to the `userService` that is done in `UserController` is mocked.
- ③ Check that the response is 200 OK.
- ④ Add the Thymeleaf resolver for the SVG images (Copied from `TamingThymeleafApplicationConfiguration` as that is not loaded automatically in a `@WebMvcTest`)
- ⑤ Add the `StubUserDetailsService` so the `@WithUserDetails` can work.

Run the test and it should be green. In the output of the test, you will see all the HTML that Thymeleaf rendered.

We have now written a test using a mock servlet environment where can be sure that a user with the appropriate authorization can access the application. However, we have not tested anything that is 'visible' in the HTML page. We could take a look at the resulting HTML body and use xpath expressions or text searches for that. But that would be extremely brittle.

A better option is using HtmlUnit so that is what we will do next.

### 15.1.3. Using HtmlUnit

To start with `HtmlUnit` and the [Spring Test HtmlUnit Integration](#), we need to add the dependency in the `pom.xml` in the `<dependencies>` section:

```

<dependency>
    <groupId>net.sourceforge.htmlunit</groupId>
    <artifactId>htmlunit</artifactId>

```

```
<scope>test</scope>
</dependency>
```

We can now again create an `@WebMvcTest` test, but we will interact with the web page under test using `com.gargoylesoftware.htmlunit.WebClient` instead of `MockMvc`.

```
package com.tamingthymeleaf.application.user.web;

import com.gargoylesoftware.htmlunit.WebClient;
import com.gargoylesoftware.htmlunit.html.*;
import
com.tamingthymeleaf.application.infrastructure.security.StubUserDetailsService;
import com.tamingthymeleaf.application.user.UserName;
import com.tamingthymeleaf.application.user.UserService;
import com.tamingthymeleaf.application.user.Users;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.context.annotation.Bean;
import org.springframework.data.domain.PageImpl;
import org.springframework.data.domain.Pageable;
import org.springframework.security.core.userdetails.UserDetailsService;
import
org.springframework.security.crypto.factory.PasswordEncoderFactories;
import org.springframework.security.crypto.password.PasswordEncoder;
import
org.springframework.security.test.context.support.WithUserDetails;
import
org.thymeleaf.spring5.templateresolver.SpringResourceTemplateResolver;
import org.thymeleaf.templateresolver.ITemplateResolver;

import java.util.List;

import static com.tamingthymeleaf.application.infrastructure.security
.StubUserDetailsService.USERNAME_ADMIN;
import static org.assertj.core.api.Assertions.assertThat;
import static org.mockito.ArgumentMatchers.any;
import static org.mockito.Mockito.when;
```

```

@WebMvcTest(UserController.class)
①
class UserControllerHtmlUnitTest {

    @Autowired
    private WebClient webClient; ②
    @MockBean
    private UserService userService;

    @BeforeEach
    void setup() { ③
        webClient.getOptions().setCssEnabled(false);
        webClient.getOptions().setJavaScriptEnabled(false);
    }

    @Test
    @WithUserDetails(USERNAME_ADMIN)
④
    void testGetUsersAsAdmin() throws Exception {
        when(userService.getUsers(any(Pageable.class))) ⑤
            .thenReturn(new
PageImpl<>(List.of(Users.createUser(new UserName("Kaden", "Whyte")),
    Users.createUser(new UserName("Charlton", "Faulkner")),
    Users.createUser(new UserName("Yuvaan", "Mcpherson"))
        )));

        HtmlPage htmlPage = webClient.getPage("/users"); ⑥
        DomNodeList<DomElement> h1headers = htmlPage
    .getElementsByTagName("h1"); ⑦
        assertThat(h1headers).hasSize(1)
            .element(0)
            .extracting(DomElement::asText)
            .isEqualTo("Users"); ⑧

        HtmlTable usersTable = htmlPage.getHtmlElementById("users-
table"); ⑨
        List<HtmlTableRow> rows = usersTable.getRows(); ⑩

        HtmlTableRow headerRow = rows.get(0); ⑪
        assertThat(headerRow.getCell(0).asText()).isEqualTo("Name");
        assertThat(headerRow.getCell(1).asText()).isEqualTo("Gender");
        assertThat(headerRow.getCell(2).asText()).isEqualTo("Birthday");
        assertThat(headerRow.getCell(3).asText()).isEqualTo("Email");
    }
}

```

```

        HtmlTableRow row1 = rows.get(1); ②
        assertThat(row1.getCell(0).asText()).isEqualTo("Kaden Whyte");
        assertThat(row1.getCell(1).asText()).isEqualTo("FEMALE");
        assertThat(row1.getCell(2).asText()).isEqualTo("2001-06-19");
        assertThat(row1.getCell(3).asText()).isEqualTo
("kaden.whYTE@gmail.com");
    }

    @TestConfiguration
    static class TestConfig {
        @Bean
        public PasswordEncoder passwordEncoder() {
            return PasswordEncoderFactories
.createDelegatingPasswordEncoder();
        }

        @Bean
        public ITemplateResolver svgTemplateResolver() {
            SpringResourceTemplateResolver resolver = new
SpringResourceTemplateResolver();
            resolver.setPrefix("classpath:/templates/svg/");
            resolver.setSuffix(".svg");
            resolver.setTemplateMode("XML");

            return resolver;
        }

        @Bean
        public UserDetailsService userDetailsService(PasswordEncoder
passwordEncoder) {
            return new StubUserDetailsService(passwordEncoder);
        }
    }
}

```

- ① Annotate the test with `@WebMvcTest` so that the mock servlet infrastructure will be set up.
- ② Inject the `WebClient` which we will use to drive `HtmlUnit`.
- ③ `HtmlUnit` does not play so well with Tailwind CSS and AlpineJS, so we disable CSS and JavaScript.
- ④ We add `@WithUserDetails` so we are using an administrator user when doing the interaction with the webpage.
- ⑤ The `UserController` interacts with the `UserService` to get the users. We setup the Mockito expectations here so we know what to expect in the HTML table that displays the information of the users.

- ⑥ Ask HtmlUnit to go to the `/users` url and return the `HtmlPage` that the browser would see.
- ⑦ Get all HTML `<h1>` tags on the page.
- ⑧ Assert that only 1 tag should be present and it should contain the text 'Users'.
- ⑨ Get the `HtmlTable` that has the `id users-table`. We updated `list.html` for this:

```
<table id="users-table" class="min-w-full">
```

- ⑩ Get all rows in the table.
- ⑪ Get the first row and assert the text of each table header.
- ⑫ Get the second row and assert the text of each table cell.



Throughout this chapter, some of the HTML elements will get an explicit `id` like we just did for the `users-table`. This makes it easy to find the elements from a test.

The book will not show every update to that, so if you are following along and a test fails, be sure to check the [GitHub sources](#) if you might be missing an id.

The test uses a helper class called `Users`:

```
package com.tamingthymeleaf.application.user;

import java.time.LocalDate;
import java.util.UUID;

public class Users {
    public static User createUser() {
        return createUser(new UserName("Henry", "Cross"));
    }

    public static User createUser(UserName userName) {
        return User.createUser(new UserId(UUID.randomUUID()),
            userName,
            "fake-encoded-password",
            Gender.FEMALE,
            LocalDate.parse("2001-06-19"),
            new Email(String.format(
                "%s.%s@gmail.com", userName.getFirstName().toLowerCase(), userName
                .getLastName().toLowerCase())),
            new PhoneNumber("+555 123 456"));
    }
}
```



Creating a helper class to create pre-populated objects for testing is a pattern I use a lot. It is called [Object Mother](#).

If I have a class `User`, I will create a class `Users` with static methods to generate test users to make the tests more readable. The naming convention is to just add `s`. So:

- `User` → `Users`
- `Product` → `Products`

Unless it already ends with an `s`, then I just append `ObjectMother`:

- `Address` → `AddressObjectMother`

We can also use `HtmlUnit` to enter form data. This test simulates an administrator clicking on the 'Add user' link, filling out all needed data and clicking on the submit button to save the new user:

```

@Test
@WithUserDetails(USERNAME_ADMIN)
void testAddUser() throws IOException {
    when(userService.getUsers(any(Pageable.class)))
        .thenReturn(Page.empty());

    HtmlPage htmlPage = webClient.getPage("/users");
    DomNodeList<DomElement> elements = htmlPage.
getElementsByTagName("a");
    Optional<DomElement> createUserLink = elements.stream()
        .filter(domElement
-> domElement.asText().equals("Add user"))
        .findFirst(); ①
    assertThat(createUserLink).isPresent();

    HtmlPage createUserFormPage = createUserLink.get().click(); ②
    assertThat(createUserFormPage).isNotNull();

    DomNodeList<DomElement> h1headers = createUserFormPage
.getElementsByTagName("h1");
    assertThat(h1headers).hasSize(1)
        .element(0)
        .extracting(DomElement::asText)
        .isEqualTo("Create user"); ③

    createUserFormPage.getElementById("gender-MALE").click();
    createUserFormPage.<HtmlTextInput>getElementByName("firstName")
.setText("John"); ⑤
    createUserFormPage.<HtmlTextInput>getElementByName("lastName")
.setText("Millen");
    createUserFormPage.<HtmlEmailInput>getElementByName("email")
.setText("john.millen@gmail.com");
    createUserFormPage.<HtmlPasswordInput>getElementByName

```

```

("password").setText("verysecure");
    createUserFormPage.<HtmlPasswordInput>getElementByName
("passwordRepeated").setText("verysecure");
    createUserFormPage.<HtmlTextInput>getElementByName(
"phoneNumber").setText("+555 123 456");
    createUserFormPage.<HtmlTextInput>getElementByName("birthday")
).setText("2004-03-27");

    HtmlPage pageAfterFormSubmit = createUserFormPage.
getElementById("submit-button").click(); ⑥
        assertThat(pageAfterFormSubmit.getUrl()).isEqualTo(new URL
("http://localhost:8080/users")); ⑦

    ArgumentCaptor<CreateUserParameters> captor = ArgumentCaptor
.forClass(CreateUserParameters.class);
    verify(userService).createUser(captor.capture()); ⑧

    CreateUserParameters parameters = captor.getValue(); ⑨
        assertThat(parameters.getUserName().getFirstName()).isEqualTo
("John"); ⑩
        assertThat(parameters.getUserName().getLastName()).isEqualTo
("Millen");
        assertThat(parameters.getEmail()).isEqualTo(new Email
("john.millen@gmail.com"));
        assertThat(parameters.getPassword()).isEqualTo("verysecure");
        assertThat(parameters.getPhoneNumber()).isEqualTo(new
PhoneNumber("+555 123 456"));
        assertThat(parameters.getBirthday()).isEqualTo(LocalDate.parse
("2004-03-27"));
    }
}

```

- ① Find the `Add user` link.
- ② Simulate a click on the link. The result of the `click()` method is the page that the browser redirects to.
- ③ Search for the `<h1>` element and assert the title text is 'Create user'.
- ④ Click on the radio button to select the `MALE` gender.
- ⑤ Find the `firstName` input and simulate entering some text in the input.
- ⑥ Simulate a click on the submit button for the form.
- ⑦ Validate that we got redirected to the `/users` URL.
- ⑧ Verify that `userService.createUser(CreateUserParameters)` method was called. By using an `ArgumentCaptor`, we can capture the `CreateUserParameters` object that was given to the mock `UserService` and verify the state of that object.
- ⑨ Get the `CreateUserParameters` object and assert each field to see if it matches with the data

that was used in the HTML form.

## 15.2. Using Cypress

In the previous section, we used [@WebMvcTest](#) to start a mock servlet environment for our testing. While this has a lot of advantages, there are also a few drawbacks:

- There might be a difference between the mock servlet environment and the actual behaviour of Tomcat.
- We are not doing an end-to-end test from HTML to database. That interaction might have some hidden bugs that we might not find by using Mockito mock services.
- Writing HtmlUnit tests is not very visual. You need to start the actual application to reference a bit what the result is of the Thymeleaf templating.
- HtmlUnit is not using an actual browser, so the execution of JavaScript and CSS might be different.

In this section, we will write an end-to-end test using [Cypress](#) to address these drawbacks. Cypress is a front end testing tool, similar to [Selenium](#). See [Cypress Features](#) for some more details about Cypress.

We will start the full application with a PostgreSQL database in Docker, the Spring Boot application locally and the Cypress test runner also in Docker. Cypress will start a browser (e.g. Chrome) and run the tests (written in JavaScript).

### 15.2.1. Cypress installation

In a later phase, we will combine the Cypress tests with JUnit so we can run them from Maven in the end. For that reason, we will follow Maven conventions and put the tests in the [src/test/e2e](#) folder.

Create a new [package.json](#) file in that directory:

`src/test/e2e/package.json`

```
{
  "name": "taming-thymeleaf-tests"
}
```

Open a terminal at [src/test/e2e](#) and run:

```
npm install cypress --save-dev
```

This will install Cypress and update the [package.json](#) file:

```
{
  "name": "taming-thymeleaf-tests",
  "devDependencies": {
    "cypress": "^5.1.0"
  }
}
```

Now run:

```
npx cypress open
```

This will start the Cypress desktop application and create example tests in [cypress/integration/examples](#):

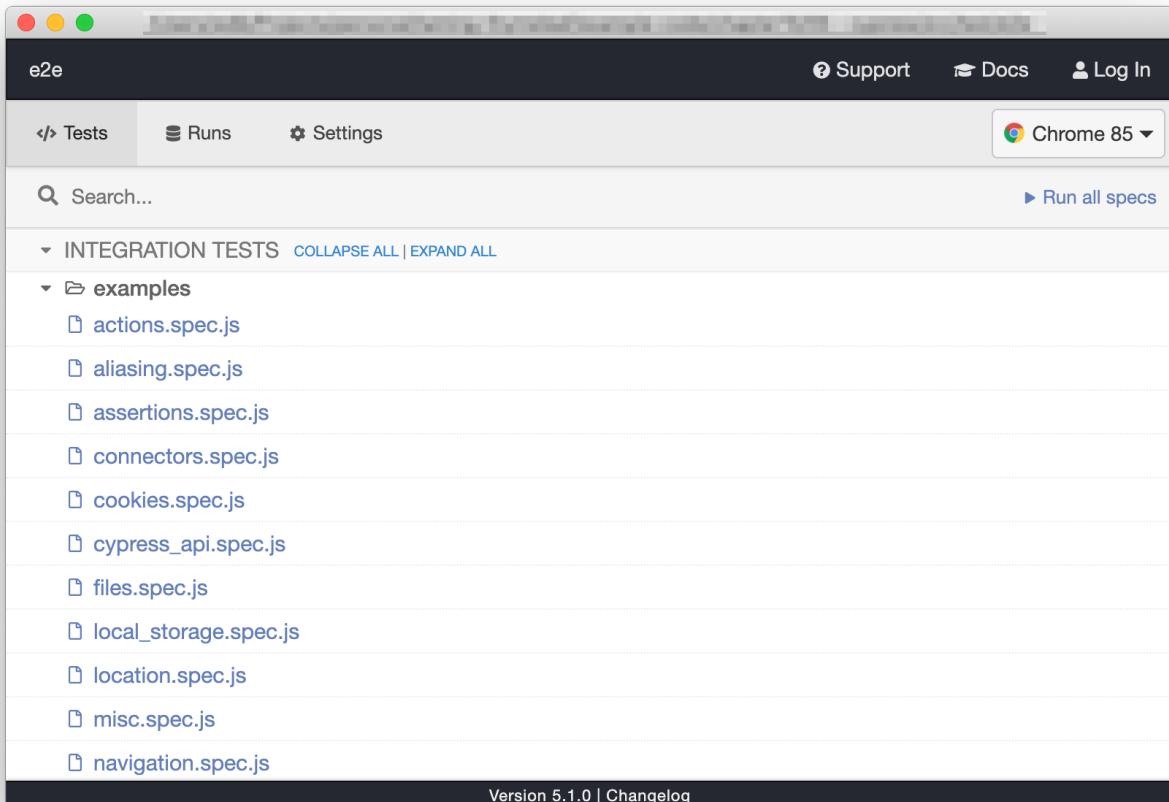


Figure 80. Cypress desktop app running on macOS

Click on [actions.spec.js](#) to see Cypress in action. It will start the browser you select in the top-right corner and run the tests.

Close Cypress again, so we can write our first test.

### 15.2.2. First Cypress test

A good unit or integration test needs to start from a well-known situation. For running the Cypress tests via JUnit, we will start our application using [@SpringBootTest](#) with an empty database. Each Cypress test will probably add something to that database. We would not want to restart the docker containers for each little Cypress test. That would just take too long.

As an alternative, we can expose certain REST API endpoints that will put the database in a well-known state. This could be completely empty, or with a few test users, or with many users to test pagination,

...

We start by adding [IntegrationTestController](#) to [src/main/java](#):

```

package com.tamingthymeleaf.application.infrastructure.test;

import com.tamingthymeleaf.application.user.*;
import org.springframework.context.annotation.Profile;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.time.LocalDate;

@RestController ①
@RequestMapping("/api/integration-test") ②
@Profile("integration-test") ③
public class IntegrationTestController {
    private final UserService userService;

    public IntegrationTestController(UserService userService) { ④
        this.userService = userService;
    }

    @PostMapping("/reset-db")
    public void resetDatabase() { ⑤
        userService.deleteAllUsers();

        addUser();
        addAdministrator();
    }

    private void addUser() {
        userService.createUser(
            new CreateUserParameters(new UserName("User", "Last"),
⑥
                "user-pwd",
                Gender.MALE,
                LocalDate.parse("2010-04-13"),
                new Email(
                    "user.last@gmail.com"),
                new PhoneNumber("+555 789
456")));
    }

    private void addAdministrator() {
        userService.createAdministrator(
            new CreateUserParameters(new UserName("Admin",

```

```

    "Strator"), ⑦
                "admin-pwd",
                Gender.FEMALE,
                LocalDate.parse("2008-09-25"),
                new Email
        ("admin.strator@gmail.com"),
                new PhoneNumber("+555 123
456"));
    }
}

```

- ① Create a `@RestController` so we can call the endpoints from the Cypress tests.
- ② All endpoints will be served at `/api/integration-test`
- ③ These endpoints should only be started when running as a test. It is very important that this is not exposed when running on production as it wipes the complete database.
- ④ Inject the `UserService` to be able to create the default users.
- ⑤ Add an endpoint `/reset-db` so the Cypress test can do a `POST` call on it.
- ⑥ Create a user with the role `USER`.
- ⑦ Create an administrator user.



We place the `IntegrationTestController` in the `main` part of the sources, not in the `test` part. The reason for this is that we will run our main application when writing the Cypress tests, with the `IntegrationTestController` enabled. This will give us a much better workflow when authoring the tests. If we would have put the controller in `src/test/java`, then this would not be possible.

Since we only start this controller when running tests, we can safely open up the security and allow everybody to access the `/api/integration-test` endpoint. This will avoid that we need to authenticate from the Cypress tests to call the endpoint:

```

@Override
protected void configure(HttpSecurity http) throws Exception {
    http.csrf(httpSecurityCsrfConfigurer ->
        httpSecurityCsrfConfigurer.
    ignoringAntMatchers("/api/integration-test/**")); ①
    http.authorizeRequests()
        .requestMatchers(PathRequest.toStaticResources
    () .atCommonLocations()).permitAll()
        .antMatchers("/api/integration-test/**").permitAll() ②
        .antMatchers("/img/*").permitAll()
        .anyRequest().authenticated()
        .and()
        .formLogin()
        .loginPage("/login")
        .permitAll()

```

```

    .and()
    .logout().permitAll();
}

```

① Disable CSRF on the `/api/integration-test` endpoints

② Allow everybody to access `/api/integration-test`

We can now write our first Cypress test. Create a new file in `src/test/e2e/cypress/integration` called `auth.spec.js` with this content:

```

/// <reference types="Cypress" />

describe('Authentication', () => { ①

  beforeEach(() => { ②
    cy.setCookie(
      'org.springframework.web.servlet.i18n.CookieLocaleResolver.LOCALE', 'en
    ); ③
    cy.request({ ④
      method: 'POST',
      url: '/api/integration-test/reset-db',
      followRedirect: false
    }).then((response) => {
      expect(response.status).to.eq(200); ⑤
    });
  });

  it('should be possible to log on as a user', () => { ⑥
    cy.visit('/login'); ⑦
  });
});

```

① `describe` is a Mocha function to declare a test suite.

② `beforeEach` (also from Mocha) allows to execute some code before each test, similar to the `@BeforeEach` annotation of JUnit 5.

③ Use `cy.setCookie` to set the language cookie that Spring Boot will read. This can be very useful to test multi-language support.

④ Using `cy.request`, we can do a REST call to our `api/integration-test/reset-db` endpoint

⑤ `expect` is a Chai assertion. We test if we got a 200 OK response from the REST call.

⑥ `it` defines an actual test.

⑦ Using `cy.visit()`, we can navigate to a particular URL.

In the test itself, we use relative URLs. We will set the base URL in the Cypress settings. Update `src/test/e2e/cypress.json` to set the base URL that all Cypress tests should use. We will also set

the default viewport that the browser will get to render the content:

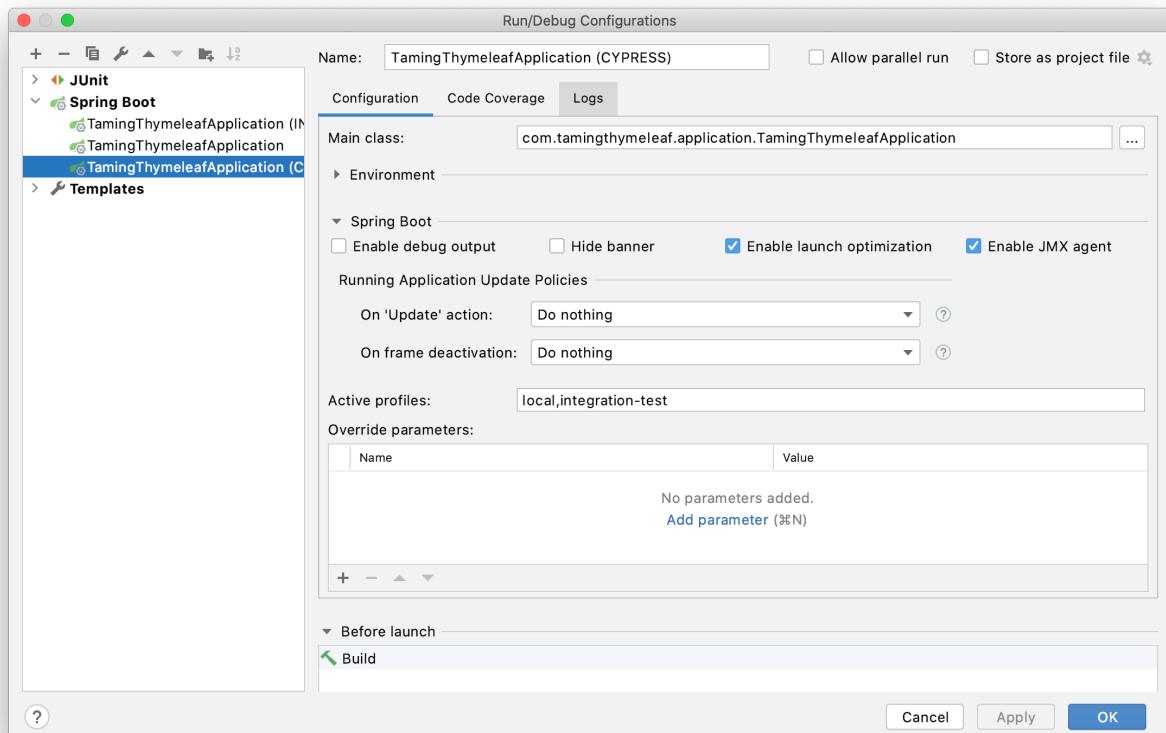
```
{
  "baseUrl": "http://localhost:8080",
  "viewportWidth": 1100,
  "viewportHeight": 800
}
```



That viewport size is only a default. It is perfectly possible to change the viewport during the test. That allows for example to check if certain elements are visible on desktop, but not on mobile.

To run the Cypress test, execute the following steps:

- Add an extra run configuration in your IDE that enables the **integration-test** profile, or ensure you add the **integration-test** profile when running from the command line:



Run the Spring Boot application with the extra profile.

- Run `npx cypress open` from the `src/test/e2e` directory.
- Click on `auth.spec.js` in the Cypress desktop application.

This will start the test and should look like this when the test is done:

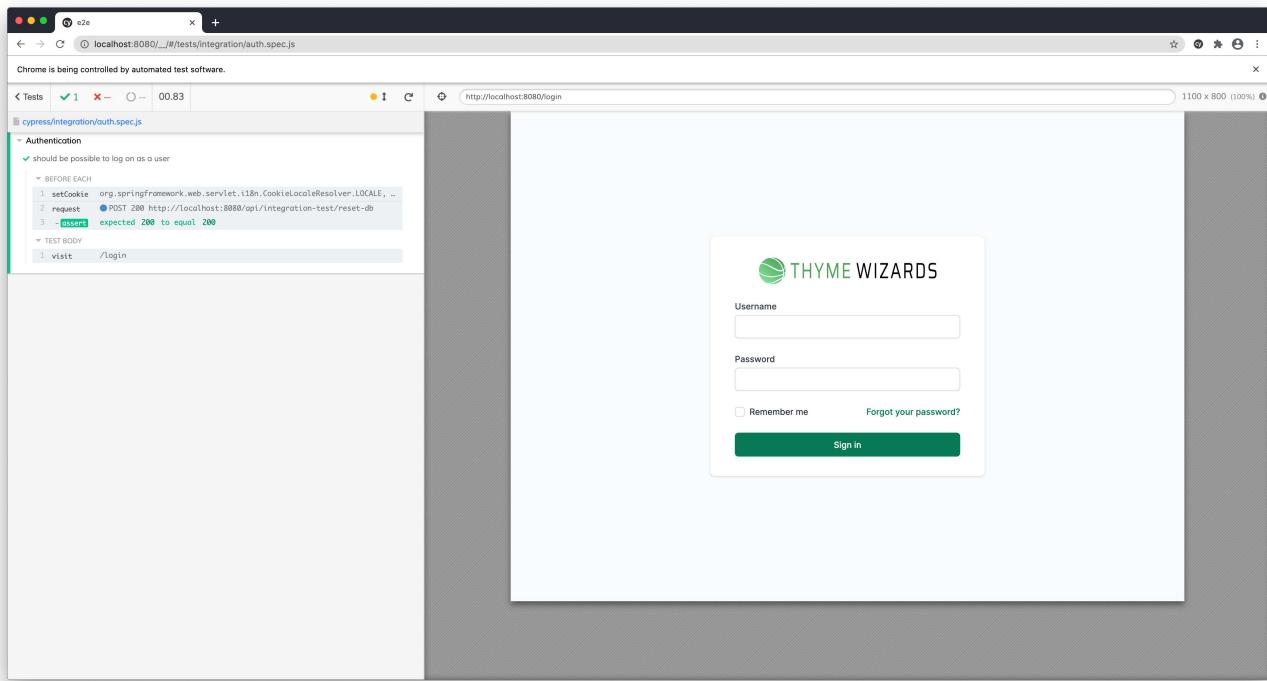


Figure 81. Cypress runner showing the test on the left and the browser contents on the right

You can now interact with our application in the Cypress controlled browser. You can for example try to log on using one of the users we created via the [IntegrationTestController](#).

We can now expand the test to actually log in and check that we get redirected to the [/users](#) URL after login:

```
it('should be possible to log on as a user', () => {
  cy.visit('/login'); ①

  cy.get('#username').type('user.last@gmail.com'); ②
  cy.get('#password').type('user-pwd'); ③
  cy.get('#submit-button').click(); ④

  cy.url().should('include', '/users'); ⑤
});
```

- ① Visit the [/login](#) page.
- ② Search for the HTML element with id `username` and type `user.last@gmail.com` in the input field.
- ③ Search for the HTML element with id `password` and type `user-pwd` in the input field.
- ④ Search for the `submit-button` HTML element and click it.
- ⑤ Assert that the URL has now changed to [/users](#).

You can just keep the Cypress desktop application open while coding the test. Each time you save, the test will re-run automatically.

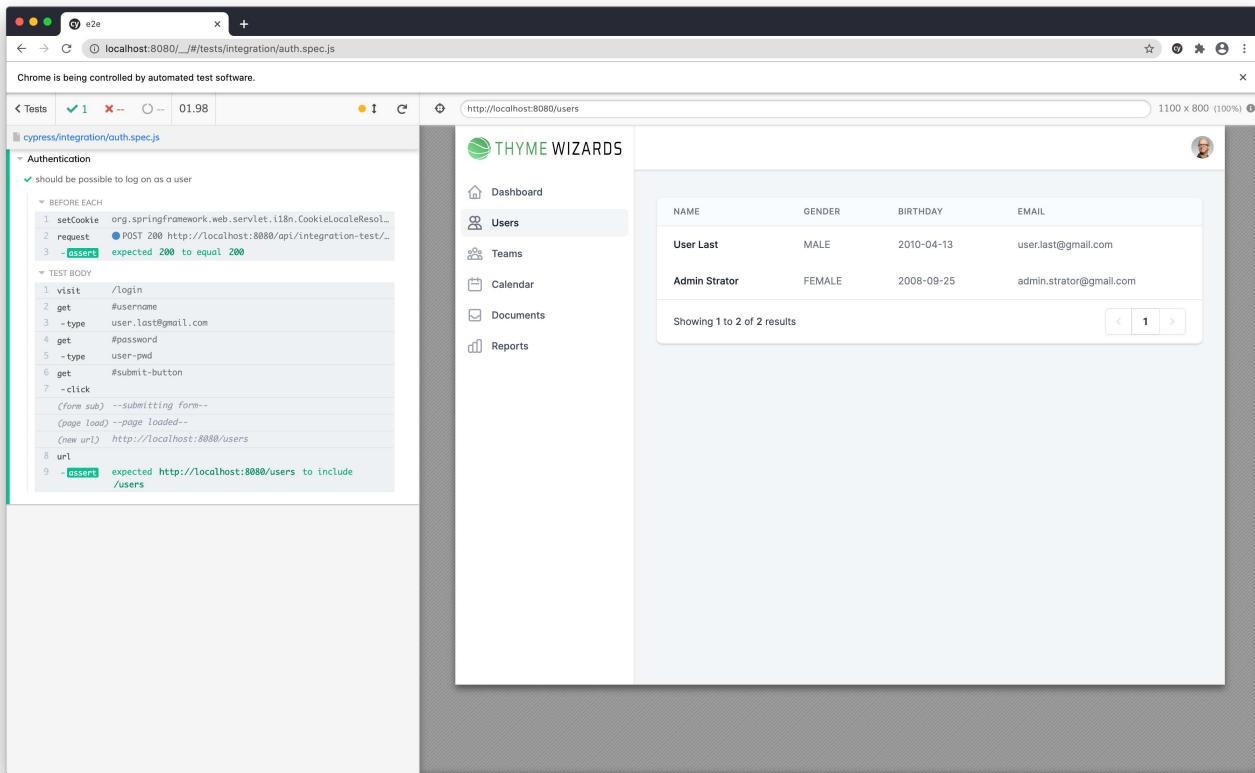


Figure 82. Result of the user login test

Let's also test the admin login:

```
it('should be possible to log on as an admin', () => {
    cy.visit('/login');

    cy.get('#username').type('admin.strator@gmail.com');
    cy.get('#password').type('admin-pwd');
    cy.get('#submit-button').click();

    cy.url().should('include', '/users');
});
```

Result:

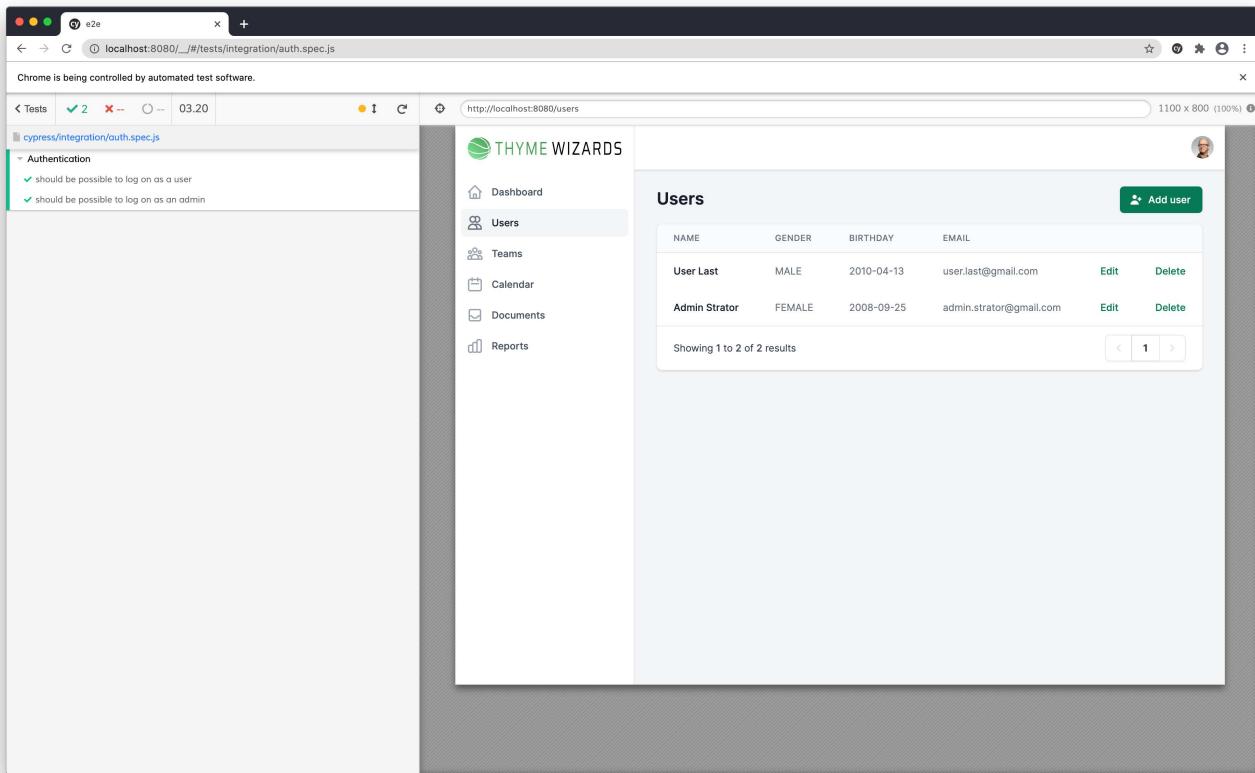


Figure 83. Result of the admin login test

### 15.2.3. Bypassing login

There is something mesmerizing about watching Cypress type and click. However, having Cypress type and click for each test to do the login before the actual test can start would be very time-consuming.

As an alternative, we will write a *Cypress command*. Cypress commands allow to create helper functions that can be used in the tests. This command will do the login by directly submitting the form data and not going to the login page and type username/email and password.

Open `src/test/e2e/cypress/support/commands.js` and add the following code:

```
Cypress.Commands.add('loginByForm', (username, password) => {
  Cypress.log({
    name: 'loginByForm',
    message: `${username} | ${password}`,
  });

  return cy.request('/login')
    .its('body')
    .then((body) => {
      // we can use Cypress.$ to parse the string body
      // thus enabling us to query into it easily
      const $html = Cypress.$(body);
      const csrf = $html.find('input[name=_csrf]').val();
```

```

        cy.loginByCSRF(username, password, csrf)
            .then((resp) => {
                expect(resp.status).to.eq(200);
            });
    });
});

Cypress.Commands.add('loginByCSRF', (username, password, csrfToken) => {
    cy.request({
        method: 'POST',
        url: '/login',
        failOnStatusCode: false, // dont fail so we can make assertions
        form: true, // we are submitting a regular form body
        body: {
            username,
            password,
            _csrf: csrfToken // insert this as part of form body
        }
    });
});

```

You don't need to know this code in detail. Just know that it adds an extra function on the `cy` object in your tests that will allow to login with:

```
cy.loginByForm('admin.strator@gmail.com', 'admin-pwd');
```

We can now write a new test in [src/test/e2e/cypress/integration/user-management.spec.js](#):

```

/// <reference types="Cypress" />

describe('User management', () => {
    beforeEach(() => {
        cy.setCookie(
            'org.springframework.web.servlet.i18n.CookieLocaleResolver.LOCALE', 'en'
        );
        cy.request({
            method: 'POST',
            url: 'api/integration-test/reset-db',
            followRedirect: false
        }).then((response) => {
            expect(response.status).to.eq(200);
        });
    });
});

```

```

    cy.loginByForm('admin.strator@gmail.com', 'admin-pwd'); ①
    cy.visit('/users'); ②
});

it('should be possible to navigate to the Add User form', () => {
    cy.get('#add-user-button').click(); ③

    cy.url().should('include', '/users/create'); ④
});
});

```

① Use the `loginByForm` function to quickly log in with the admin user

② Go to the `/users` URL of the application.

③ Find the `add-user-button` and click on it.

④ Assert that we arrived at the `/users/create` URL.

With all this in place, we have a framework for writing tests for all the functionality we have added to our application already. In the next section, we will integrate the Cypress tests with the other automated tests.

#### 15.2.4. Running Cypress tests from JUnit

Now that we have a few Cypress tests, we need to find a way to run them automatically when we build our project. The easiest way to do that is run them from JUnit, together with our other automated tests.

To avoid that we need to install Cypress for running the tests, we will use Testcontainers with a Docker image that contains Cypress.

Start by adding 2 dependencies to the `pom.xml`:

```

<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>${testcontainers.version}</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>io.github.wimdeblauwe</groupId>
    <artifactId>testcontainers-cypress</artifactId>
    <version>${testcontainers-cypress.version}</version>
    <scope>test</scope>
</dependency>

```

The `org.testcontainers:junit-jupiter` dependency allows to use Testcontainers from JUnit 5. The `io.github.wimdeblauwe:testcontainers-cypress` dependency allows to use the Cypress

docker image from Testcontainers.

We will start with an integration test that starts PostgreSQL in Docker and the full application locally on a random port:

```
package com.tamingthymeleaf.application;

import com.tamingthymeleaf.application.user.UserService;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.web.server.LocalServerPort;
import org.springframework.test.context.DynamicPropertyRegistry;
import org.springframework.test.context.DynamicPropertySource;
import org.testcontainers.containers.PostgreSQLContainer;
import org.testcontainers.junit.jupiter.Container;
import org.testcontainers.junit.jupiter.Testcontainers;

import static org.assertj.core.api.Assertions.assertThat;

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment
.RANDOM_PORT) ①
@Testcontainers ②
public class CypressE2eTests {
    @Container ③
    private static final PostgreSQLContainer postgresqlContainer = new
PostgreSQLContainer<>("postgres:12")
        .withDatabaseName("tamingthymeleafdb")
        .withUsername("user")
        .withPassword("secret");

    @LocalServerPort ④
    private int port;

    @Autowired
    private UserService userService; ⑤

    @DynamicPropertySource
    static void setup(DynamicPropertyRegistry registry) { ⑥
        registry.add("spring.datasource.url", postgresqlContainer:
:getJdbcUrl);
        registry.add("spring.datasource.username", postgresqlContainer:
:getName());
        registry.add("spring.datasource.password", postgresqlContainer:

```

```

    :getPassword());
}

@BeforeEach
void validatePreconditions() {
    assertThat(userService.countUsers()).isZero(); ⑦
}

@Test
void test() {
    System.out.println("port = " + port);
    System.out.println("Application started");
}

}

```

- ① `@SpringBootTest` starts the complete application. By setting the `webEnvironment` to `RANDOM_PORT` an embedded Tomcat is started on a random port.
- ② The `@Testcontainers` annotation triggers the JUnit 5 support of Testcontainers.
- ③ The `@Container` annotation will start and stop the Docker container via Testcontainers at the appropriate points in the JUnit lifecycle. By default, `PostgreSQLContainer` uses version `9.6.12`, but by specifying the name of the docker image, we can use a more recent version. See [Postgres on Docker Hub](#) for all available versions.
- ④ Because the application starts at a random port, we will need to know what port that is so we can point Cypress to the good URL. Using `@LocalServerPort`, Spring will inject the chosen port into the `port` variable.
- ⑤ We can autowire any bean from the Spring context in our test if needed.
- ⑥ We can dynamically set the JDBC URL, database username and password by adding this method annotated with `@DynamicPropertySource`. It is an easy way to pass the information from the `PostgreSQLContainer` to our application.
- ⑦ The database should be empty at startup.

Run this test. It should start PostgreSQL and the application, print the port and shutdown again.

Building upon this test, we can create the full integration between Cypress and JUnit.

1. Remove the `src/test/e2e/cypress/integration/examples` directory as we don't need the examples.
2. Run (in the `src/test/e2e` directory):

```
npm install cypress-multi-reporters mocha mochawesome --save-dev
```

This adds the mochawesome reporter to the project which generates test results in JSON format. `testcontainers-cypress` will read those reports to report the results back to JUnit.

3. Update `src/test/e2e/cypress.json`:

```
{
  "baseUrl": "http://localhost:8080",
  "viewportWidth": 1100,
  "viewportHeight": 800,
  "reporter": "cypress-multi-reporters",
  "reporterOptions": {
    "configFile": "reporter-config.json"
  }
}
```

4. Create a `reporter-config.json` file (next to `cypress.json`):

```
{
  "reporterEnabled": "spec, mochawesome",
  "mochawesomeReporterOptions": {
    "reportDir": "cypress/reports/mochawesome",
    "overwrite": false,
    "html": false,
    "json": true
  }
}
```

5. Add the following to `.gitignore` to avoid accidental commits:

```
src/test/e2e/cypress/reports
src/test/e2e/cypress/videos
src/test/e2e/cypress/screenshots
```

6. Update the Maven `pom.xml` so the Cypress tests are made available as test resources:

```
<project>
  <build>
    ...
    <testResources>
      <testResource>
        <directory>src/test/resources</directory>
      </testResource>
      <testResource>
        <directory>src/test/e2e</directory>
        <targetPath>e2e</targetPath>
      </testResource>
    </testResources>
```

```

    ...
</build>
</project>
```

Now, we will start the Cypress Docker container in our JUnit test and run the Cypress tests:

```
package com.tamingthymeleaf.application;

import com.tamingthymeleaf.application.user.UserService;
import io.github.wimdeblauwe.testcontainers.cypress.CypressContainer;
import io.github.wimdeblauwe.testcontainers.cypress.CypressTestResults;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.web.server.LocalServerPort;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.context.DynamicPropertyRegistry;
import org.springframework.test.context.DynamicPropertySource;
import org.testcontainers.containers.PostgreSQLContainer;
import org.testcontainers.junit.jupiter.Container;
import org.testcontainers.junit.jupiter.Testcontainers;

import java.io.IOException;
import java.util.concurrent.TimeoutException;

import static org.assertj.core.api.Assertions.assertThat;

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment
.RANDOM_PORT)
@Testcontainers
@ActiveProfiles("integration-test") ①
public class CypressE2eTests {
    @Container
    private static final PostgreSQLContainer postgresqlContainer = new
PostgreSQLContainer<>("postgres:12")
        .withDatabaseName("tamingthymeleafdb")
        .withUsername("user")
        .withPassword("secret");

    @LocalServerPort
    private int port;

    @Autowired
```

```

private UserService userService;

@DynamicPropertySource
static void setup(DynamicPropertyRegistry registry) {
    registry.add("spring.datasource.url", postgresqlContainer:
:jdbcUrl);
    registry.add("spring.datasource.username", postgresqlContainer:
:username);
    registry.add("spring.datasource.password", postgresqlContainer:
:password);
}

@BeforeEach
void validatePreconditions() {
    assertThat(userService.countUsers()).isZero();
}

@Test
void runTests() throws InterruptedException, IOException,
TimeoutException {
    // Ensure that the Cypress container can access the Spring Boot
    app running on port `port` via `host.testcontainers.internal`
    org.testcontainers.Testcontainers.exposeHostPorts(port); ②
    try (CypressContainer container = new CypressContainer() ③

.withLocalServerPort(port)) { ④
        container.start(); ⑤
        CypressTestResults testResults = container.getTestResults();
⑥

        assertThat(testResults.getNumberOfFailingTests()) ⑦

.describedAs("%s", testResults)
            .isZero();
    }
}
}

```

- ① Ensure the `integration-test` profile is active so our `/api/integration-test` REST endpoint will be available.
- ② Ensure that the Cypress container can access the Spring Boot app running on port `port` via `host.testcontainers.internal`. This is a special hostname that allows a Testcontainers container to access the host the container is running on. The `testcontainers-cypress` library uses `http://host.testcontainers.internal` as default base URL.

- ③ Declare a [CypressContainer](#) with a custom Docker image name so we can match the Cypress version with the one we have been using before.
- ④ Pass the random [port](#) that Spring Boot started on to [CypressContainer](#) so the base URL for the Cypress tests can be set correctly.
- ⑤ Start the container.
- ⑥ Wait for the tests to finish and get the results.
- ⑦ Assert that there should be no failing tests.

If you like to see the output from the Cypress container, add the following to [application-integration-test.properties](#):

`src/test/resources/application-integration-test.properties`

```
logging.level.io.github.wimdeblauwe.testcontainers=DEBUG
```

If all goes well, the Cypress tests should run and the test should succeed.

Cypress automatically creates videos of each test. You can view this video at [target/test-classes/e2e/cypress/videos](#).

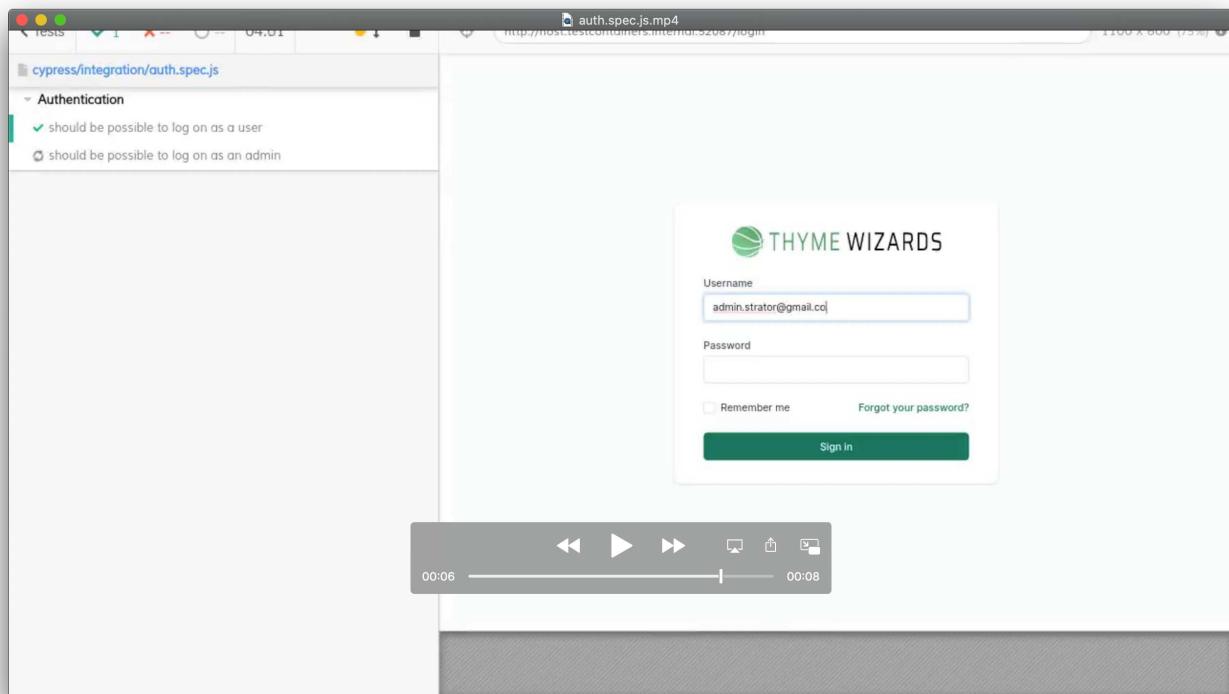


Figure 84. Video created by Cypress during testing

### 15.2.5. JUnit Testfactory

When we run the [CypressE2eTests](#) from our IDE, we only see 1 test:



This is because the tests run in the Cypress Docker container and we only get the results at the end of the full test run.

It would be a lot better if we could see the Cypress tests individually, so we can know exactly what test failed. This is possible by using the JUnit 5 support for dynamic tests:

```
package com.tamingthymeleaf.application;

import com.tamingthymeleaf.application.user.UserService;
import io.github.wimdeblauwe.testcontainers.cypress.CypressContainer;
import io.github.wimdeblauwe.testcontainers.cypress.CypressTest;
import io.github.wimdeblauwe.testcontainers.cypress.CypressTestResults;
import io.github.wimdeblauwe.testcontainers.cypress.CypressTestSuite;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DynamicContainer;
import org.junit.jupiter.api.DynamicTest;
import org.junit.jupiter.api.TestFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.web.server.LocalServerPort;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.context.DynamicPropertyRegistry;
import org.springframework.test.context.DynamicPropertySource;
import org.testcontainers.containers.PostgreSQLContainer;
import org.testcontainers.junit.jupiter.Container;
import org.testcontainers.junit.jupiter.Testcontainers;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.TimeoutException;

import static org.assertj.core.api.Assertions.assertThat;
import static org.junit.jupiter.api.Assertions.assertTrue;
```

```

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment
.RANDOM_PORT)
@Testcontainers
@ActiveProfiles("integration-test")
public class CypressE2eTests {
    @Container
    private static final PostgreSQLContainer postgresqlContainer = new
PostgreSQLContainer<>("postgres:12")
        .withDatabaseName("tamingthymeleafdb")
        .withUsername("user")
        .withPassword("secret");

    @LocalServerPort
    private int port;

    @Autowired
    private UserService userService;

    @DynamicPropertySource
    static void setup(DynamicPropertyRegistry registry) {
        registry.add("spring.datasource.url", postgresqlContainer:
:getJdbcUrl);
        registry.add("spring.datasource.username", postgresqlContainer:
:getUsername);
        registry.add("spring.datasource.password", postgresqlContainer:
: getPassword);
    }

    @BeforeEach
    void validatePreconditions() {
        assertThat(userService.countUsers()).isZero();
    }

    @TestFactory ①
    List<DynamicContainer> runTests() throws InterruptedException,
IOException, TimeoutException { ②
        // Ensure that the Cypress container can access the Spring Boot
app running on port `port` via `host.testcontainers.internal`
        org.testcontainers.Testcontainers.exposeHostPorts(port);
        try (CypressContainer container = new CypressContainer()
            .withLocalServerPort(port)) {
            container.start();
            CypressTestResults testResults = container.getTestResults();

            return convertToJUnitDynamicTests(testResults); ③
    }
}

```

```

    }

}

private List<DynamicContainer> convertToJUnitDynamicTests
(CypressTestResults testResults) {
    List<DynamicContainer> dynamicContainers = new ArrayList<>();
    List<CypressTestSuite> suites = testResults.getSuites();
    for (CypressTestSuite suite : suites) {
        createContainerFromSuite(dynamicContainers, suite);
    }
    return dynamicContainers;
}

private void createContainerFromSuite(List<DynamicContainer>
dynamicContainers, CypressTestSuite suite) {
    List<DynamicTest> dynamicTests = new ArrayList<>();
    for (CypressTest test : suite.getTests()) {
        dynamicTests.add(DynamicTest.dynamicTest(test.
getDescription(), () -> assertTrue(test.isSuccess())));
    }
    dynamicContainers.add(DynamicContainer.dynamicContainer(suite
.getTitle(), dynamicTests));
}
}

```

- ① Replace `@Test` with `@TestFactory` to indicate that this method will return a collection of tests.
- ② Change the return type from `void` to `List<DynamicContainer>`.
- ③ Convert the test results from the `CypressContainer` to a `List<DynamicContainer>` using the `convertToJUnitDynamicTests` and `createContainerFromSuite` helper functions.

If we now run the test again from our IDE, we see in the end a nice overview of all individual tests:

The screenshot shows the IntelliJ IDEA Run tool window with the following details:

- Run Configuration:** CypressE2eTests
- Tree View (Left):**
  - Test Results
  - CypressE2eTests
    - runTests()
      - Authentication
        - should be possible to log on as a user
        - should be possible to log on as an admin
      - User management
        - should be possible to navigate to the Admin page- Log View (Right):**
  - Tests passed: 3 of 3 tests – 4 ms
  - 2020-09-08 11:01:55.137 DEBUG 40608 --- [ream-1501477292] i.g.w.t.cypress.CypressContainer
  - 2020-09-08 11:01:55.137 DEBUG 40608 --- [ream-1501477292] i.g.w.t.cypress.CypressContainer
  - 2020-09-08 11:01:55.137 DEBUG 40608 --- [ream-1501477292] i.g.w.t.cypress.CypressContainer
  - 2020-09-08 11:01:55.198 INFO 40608 --- [main] i.g.w.t.cypress.CypressContainer
  - Cypress tests passing: 3
  - Cypress tests failing: 0
  - 2020-09-08 11:01:55.297 DEBUG 40608 --- [ream-1501477292] i.g.w.t.cypress.CypressContainer
  - 2020-09-08 11:01:55.735 INFO 40608 --- [extShutdownHook] j.LocalContainerEntityManagerFactoryBean : Closing JPA EntityManagerFactory
  - 2020-09-08 11:01:55.736 INFO 40608 --- [extShutdownHook] o.s.s.concurrent.ThreadPoolTaskExecutor : Shutting down ExecutorService
  - 2020-09-08 11:01:55.736 INFO 40608 --- [extShutdownHook] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Shutdown initiated
  - 2020-09-08 11:01:55.740 INFO 40608 --- [extShutdownHook] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Shutdown completed
  - Process finished with exit code 0

This concludes the section on testing. To learn more about Cypress, check out the excellent [Cypress Documentation](#).

## 15.3. Summary

In this chapter, you learned:

- How to write a test with [@WebMvcTest](#) and HtmlUnit.
- How to simulate an authenticated user with [@WebMvcTest](#).
- How to use Cypress to write tests
- How to run Cypress tests from JUnit to integrate them in the Maven lifecycle.



If you want to learn more about testing with Spring Boot, be sure to check out the excellent [Testing Spring Boot Applications Masterclass](#) by Philip Riecks.

# Chapter 16. Various tips and tricks

## 16.1. Open Session In View

### 16.1.1. What is it?

If you studied the logging output of the application, you might have already seen this warning:

```
2020-09-08 17:06:32.095  WARN 75541 --- [           main]
JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is
enabled by default. Therefore, database queries may be performed during
view rendering. Explicitly configure spring.jpa.open-in-view to disable
this warning
```

By default, Spring Boot has 'Open Session In View' enabled. However, this is considered to be an anti-pattern by many. See [The Open Session In View Anti-Pattern](#) and [Open session in view is evil](#).

Why is that?

To answer that question, let's first explain what Open Session In View does:

Put simple: The 'Session' is what JPA/Hibernate needs to do the database work. When a controller calls a service, a transaction is started and spans all database calls that the service does (via a repository). When the service returns the deserialized objects to the controller, the transaction/session is closed.

As a performance optimization, some references of a returned entity are lazy loaded. So only if you call the getter method, the actual values are queried from the database. However, if you call this getter in the controller, there is a problem. Hibernate wants to do a database query for the additional information, but there is no session. As a result, it will throw a [LazyInitializationException](#).

To avoid that [LazyInitializationException](#), the Open Session In View pattern was invented. The session is kept open for the controller, so if Hibernate wants to do an additional query, it can do so without the exception.

So why is that bad?

It is bad because:

- The transaction is closed at the service layer. The additional statements are done under auto-commit, which causes a lot of I/O pressure on the database.
- The controller triggers 'hidden' queries which might lead to [N+1 query problems](#).
- The database connection is held longer than necessary, so the overall throughput is limited.

For all of these reasons, I turn off Open Session In View in my applications by adding this line to [application.properties](#):

```
spring.jpa.open-in-view=false
```

### 16.1.2. Consequences of disabling

Now that we disabled Open Session In View, how will we avoid that dreaded [LazyInitializationException](#) happening?

You could make all associations Eager, but then you fetch too much data all the time. It is better to make associations Lazy by default and use [join fetch](#) when reading the data.

A few tips:

- [@OneToMany](#) is already Lazy by default, so you can use that as is.
- [@ManyToOne](#) is Eager by default, so use [@ManyToOne\(fetch = FetchType.LAZY\)](#)
- Use JOIN FETCH when information from the associations is needed (If you get a [LazyInitializationException](#), it is needed).

Example of a [User](#) entity that has an association with a [Set](#) of [Vehicle](#) entities:

```
@Entity
public class User {

    @OneToMany(mappedBy = "user", cascade = CascadeType.ALL,
    orphanRemoval = true) ①
    private final Set<Vehicle> vehicles;
```

① [@OneToOne](#) is LAZY by default

Example of using [JOIN FETCH](#) in a JPQL query:

```
@Query("SELECT u FROM User u LEFT JOIN FETCH u.vehicles WHERE u.id =
:id")
Optional<User> findUserWithVehiclesById(@Param("id") UserId userId);
```

This will get the matching user with the vehicles set fully initialized in a single query, thus avoiding the [N+1 query problem](#).

To end, I want to point out that for many small applications, having OSIV enabled might not be a performance problem at all. So don't feel bad if you want to keep using it, just be aware of what it does exactly, and the trade-offs that are there.

## 16.2. StringTrimmerEditor

Users of your application might add one or more extra spaces when they need to input some data. You could ensure to trim that in each of the form data backing objects, but that would get tedious fast.

An easier way, is using [org.springframework.beans.propertyeditors.StringTrimmerEditor](#).

Add this to `GlobalControllerAdvice` so it is enabled for all controllers in the application:

```
package com.tamingthymeleaf.application.infrastructure.web;

import org.springframework.beans.propertyeditors.StringTrimmerEditor;
import org.springframework.dao.DataIntegrityViolationException;
import org.springframework.http.HttpStatus;
import org.springframework.orm.ObjectOptimisticLockingFailureException;
import org.springframework.web.bind.WebDataBinder;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.InitBinder;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.servlet.ModelAndView;

import javax.servlet.http.HttpServletRequest;

@ControllerAdvice
public class GlobalControllerAdvice {

    @ResponseStatus(HttpStatus.CONFLICT)
    @ExceptionHandler({DataIntegrityViolationException.class,
ObjectOptimisticLockingFailureException.class})
    public ModelAndView handleConflict(HttpServletRequest request,
Exception e) {
        ModelAndView result = new ModelAndView("error/409");
        result.addObject("url", request.getRequestURL());
        return result;
    }

    @InitBinder ①
    public void initBinder(WebDataBinder binder) {
        StringTrimmerEditor stringtrimmer = new StringTrimmerEditor
(false); ②
        binder.registerCustomEditor(String.class, stringtrimmer); ③
    }
}
```

① Methods annotated with `@InitBinder` will be called by the framework to initialize the `WebDataBinder`.

② Create a `StringTrimmerEditor` instance. The boolean flag indicates if you want to have an empty string returned as `null` (use `true`), or if an empty string should remain an empty string (use `false`).

③ Register the `StringTimmerEditor` to the binder for all fields of type `String`.

Now all excess whitespace will be trimmed when the values are taken from the <input> fields and put in the form data object.

## 16.3. Global model attributes

### 16.3.1. Controller specific

If we look at the various methods in [UserController](#), we see that some of the model attributes are added to the model in each method.

Let's look at a single method for reference:

```
@GetMapping("/create")
@Secured("ROLE_ADMIN")
public String createUserForm(Model model) {
    model.addAttribute("user", new CreateUserFormData());
    model.addAttribute("genders", List.of(Gender.MALE, Gender.
FEMALE, Gender.OTHER));
    model.addAttribute("possibleRoles", List.of(UserRole.values()));
    model.addAttribute("EditMode", EditMode.CREATE);
    return "users/edit";
}
```

The `genders` and `possibleRoles` attribute is also added in 3 other methods of the [UserController](#).

We could of course, just create a method and call that method from all places, but there is also an other way: add a method annotated with `@ModelAttribute` in [UserController](#).

*com.tamingthymeleaf.application.user.web.UserController*

```
@ModelAttribute("genders")
public List<Gender> genders() {
    return List.of(Gender.MALE, Gender.FEMALE, Gender.OTHER);
}

@ModelAttribute("possibleRoles")
public List<UserRole> possibleRoles() {
    return List.of(UserRole.values());
}
```

We can now remove adding those attributes to the Model in the actual controller methods:

*com.tamingthymeleaf.application.user.web.UserController*

```
@GetMapping("/create")
```

```

@Secured("ROLE_ADMIN")
public String createUserForm(Model model) {
    model.addAttribute("user", new CreateUserFormData());
    model.addAttribute("editMode",EditMode.CREATE);
    return "users/edit";
}

```

### 16.3.2. Application wide

If we have a model attribute that is needed across the whole application, we can use a `@ControllerAdvice`.

An example is adding a footer with the application version on each page of the application.

Add an extra method annotated with `@ModelAttribute` in `GlobalControllerAdvice`:

```

@ControllerAdvice
public class GlobalControllerAdvice {

    @Value("${application.version}") ①
    private String version;

    @ModelAttribute("version") ②
    public String getVersion() {
        return version;
    }
    ...
}

```

① Read the `application.version` property and inject it into the `version` field.

② Declare the result of the method call as the `version` model attribute.

In this example, we will just put the `application.version` property in the `application.properties` file:

`src/main/resources/application.properties`

```
application.version=1.0.0-SNAPSHOT
```

If you want this to be the Maven version automatically, then use:

`src/main/resources/application.properties`

```
application.version=@project.version@
```



And configure Maven resource filtering to replace the `@project.version@` with the

actual Maven version of the project:

```
<project>
  <build>
    <resources>
      <resource>
        <directory>src/main/resources</directory>
        <filtering>true</filtering>
        <includes>
          <include>
            application.properties</include>
          </includes>
        </resource>
      </resources>
    </build>
  </project>
```

We can now use `version` in all Thymeleaf templates. For example, add this to `login.html`:

```
<div class="text-xs mt-2 text-gray-500 text-center">
  <span th:text="${version}"></span>
</div>
```

Which renders in the browser as:

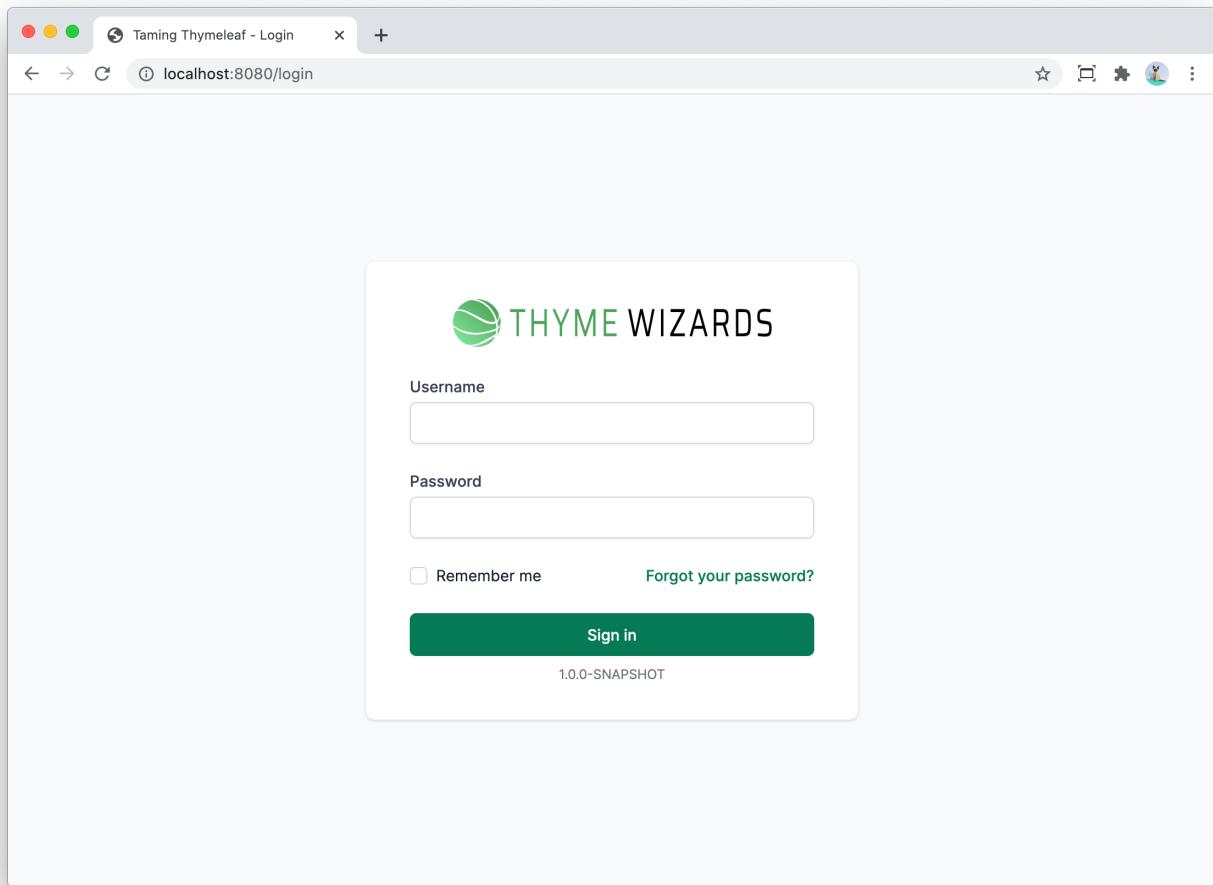


Figure 85. Login page showing version in the footer

## 16.4. File upload

There are many applications that need file upload in one form or another. For example, to add attachments to something, or to upload an avatar for a user. Let's add avatars for our users in the example application to show how file upload can be implemented.

We start by adding an `org.springframework.web.multipart.MultipartFile` field to `AbstractUserFormData`:

```
public class AbstractUserFormData {  
    ...  
    private MultipartFile avatarFile;  
  
    public MultipartFile getAvatarFile() {  
        return avatarFile;  
    }  
  
    public void setAvatarFile(MultipartFile avatarFile) {  
        this.avatarFile = avatarFile;  
    }  
}
```

```
}
```

This will allow us to map a selected file in an `<input type="file">` from the `<form>` to the `avatarFile` field.

Next, we update `CreateUserParameters` to also add a `MultipartFile` field:

`com.tamingthymeleaf.application.user.CreateUserParameters`

```
public class CreateUserParameters {
    ...
    private MultipartFile avatar;

    @Nullable
    public MultipartFile getAvatar() {
        return avatar;
    }

    public void setAvatar(MultipartFile avatar) {
        this.avatar = avatar;
    }
}
```

When converting `CreateUserFormData` to `CreateUserParameters`, we take the `avatar` field into account:

`com.tamingthymeleaf.application.user.web.CreateUserFormData`

```
public CreateUserParameters toParameters() {
    CreateUserParameters parameters = new CreateUserParameters(new
    UserName(getFirstName(), getLastName()),

    password,

    getGender(),

    getBirthday(),

    new
    Email(getEmail()),

    new
    PhoneNumber(getPhoneNumber()));

    if (getAvatarFile() != null
        && !getAvatarFile().isEmpty()) { ①
        parameters.setAvatar(getAvatarFile());
    }
}
```

```

        return parameters;
    }
}
```

- ① If the form data has a valid `MultipartFile`, pass it to the `CreateUserParameters` object.

Same for `EditUserData`:

`com.tamingthymeleaf.application.user.web.EditUserData`

```

public EditUserParameters toParameters() {
    EditUserParameters parameters = new EditUserParameters(version,
        new
    UserName(getFirstName(), getLastname()),
    getGender(),
    getBirthday(),
        new
    Email(getEmail()),
        new
    PhoneNumber(getPhoneNumber()));

    if (getAvatarFile() != null
        && !getAvatarFile().isEmpty()) {
        parameters.setAvatar(getAvatarFile());
    }

    return parameters;
}
```

We will store the image into the database. We need to update the `User` entity for this:

```

public class User extends AbstractVersionedEntity<UserId> {
    ...
    private byte[] avatar;

    /**
     * The avatar image of the driver. Null if no avatar has been set.
     *
     * @return the image bytes
     */
    public byte[] getAvatar() {
        return avatar;
    }
}
```

```

/**
 * Set the avatar image of the driver.
 *
 * @param avatar the image bytes
 */
public void setAvatar(byte[] avatar) {
    this.avatar = avatar;
}
}

```

We also need to change the Flyway scripts to allow storing the avatar `byte[]`:

```

CREATE TABLE tt_user
(
    id          UUID      NOT NULL,
    version     BIGINT    NOT NULL,
    password    VARCHAR   NOT NULL,
    first_name  VARCHAR   NOT NULL,
    last_name   VARCHAR   NOT NULL,
    gender      VARCHAR   NOT NULL,
    birthday    DATE      NOT NULL,
    email       VARCHAR   NOT NULL,
    phone_number VARCHAR  NOT NULL,
    avatar      BYTEA, ①
    PRIMARY KEY (id)
);

```

① `avatar` column to store the image

Next, update `UserServiceImpl` to take the `MultipartFile` and store the bytes in the database:

`com.tamingthymeleaf.application.user.UserServiceImpl`

```

@Override
public User createUser(CreateUserParameters parameters) {
    LOGGER.debug("Creating user {} ({})", parameters.getUserName()
    .getFullName(), parameters.getEmail().asString());
    UserId userId = repository.nextId();
    String encodedPassword = passwordEncoder.encode(parameters
    .getPassword());
    User user = User.createUser(userId,
        parameters.getUserName(),
        encodedPassword,
        parameters.getGender(),

```

```

        parameters.getBirthday(),
        parameters.getEmail(),
        parameters.getPhoneNumber());
    storeAvatarIfPresent(parameters, user); ①
    return repository.save(user);
}

private void storeAvatarIfPresent(CreateUserParameters parameters,
User user) {
    MultipartFile avatar = parameters.getAvatar(); ②
    if (avatar != null) {
        try {
            user.setAvatar(avatar.getBytes()); ③
        } catch (IOException e) {
            throw new UserServiceException(e);
        }
    }
}

```

① Use the private helper method `storeAvatarIfPresent`.

② Get the `MultipartFile` from the parameters.

③ Get the bytes from the `MultipartFile` and store them in the `User` entity.



If you want to resize the selected file before storing it, you can use the `Thumbnailator` library.

For the edit flow, we change the `update` method in `EditUserParameters`:

```

public void update(User user) {
    user.setUserName(getUserName());
    user.setGender(getGender());
    user.setBirthday(getBirthday());
    user.setEmail(getEmail());
    user.setPhoneNumber(getPhoneNumber());

    MultipartFile avatar = getAvatar();
    if (avatar != null) {
        try {
            user.setAvatar(avatar.getBytes());
        } catch (IOException e) {
            throw new UserServiceException(e);
        }
    }
}

```

With all this Java code in place, we can update `edit.html` to allow the user to select a file. We will also support showing the current image from the database when editing a user. To make that possible, we need do one last change to the Java code:

```
package com.tamingthymeleaf.application.user.web;

import com.tamingthymeleaf.application.user.*;
import java.util.Base64;

public class EditUserFormData extends AbstractUserFormData {
    private String id;
    private long version;
    private String avatarBase64Encoded; ①

    public static EditUserFormData fromUser(User user) {
        EditUserFormData result = new EditUserFormData();
        result.setId(user.getId().asString());
        result.setVersion(user.getVersion());
        result.setFirstName(user.getUserName().getFirstName());
        result.setLastName(user.getUserName().getLastName());
        result.setGender(user.getGender());
        result.setBirthday(user.getBirthday());
        result.setEmail(user.getEmail().asString());
        result.setPhoneNumber(user.getPhoneNumber().asString());

        if (user.getAvatar() != null) {
            String encoded = Base64.getEncoder().encodeToString(user
                .getAvatar()); ②
            result.setAvatarBase64Encoded(encoded);
        }
        return result;
    }

    ...

    public String getAvatarBase64Encoded() {
        return avatarBase64Encoded;
    }

    public void setAvatarBase64Encoded(String avatarBase64Encoded) {
        this.avatarBase64Encoded = avatarBase64Encoded;
    }
}
```

- ① The `avatarBase64Encoded` field will contain the avatar in Base64 encoding so we can display it using an `<img>` tag.
- ② Convert the `byte[]` from the avatar to a Base64 String.

Now onto `edit.html`. This is the `<div>` that has the relevant code for adding or editing an avatar:

```

<div class="sm:col-span-6 flex flex-col items-center sm:flex-row sm:justify-start">
     ①
    <input id="avatarFile" type="file" name="avatarFile" class="hidden">
    ②
    <button id="selectAvatarButton"
        type="button"
        class="ml-4 py-2 px-3 border border-gray-300 rounded-md
text-sm leading-4 font-medium text-gray-700 hover:text-gray-500
focus:outline-none focus:border-blue-300 focus:shadow-outline-blue
active:bg-gray-50 active:text-gray-800"
        th:text="#{user.avatar.add}">Add picture
    </button> ③
</div>

```

- ① The `<img>` tag will either show the current avatar of the user, or it will show a placeholder SVG.
- ② This is the `<input>` that is mapped on the `MultipartFile` field of the form data objects. Note that the `name` needs to match with the name of the field in the Java objects. We make the input `hidden` because we don't want to use the standard file upload button in this example.
- ③ The button is there to allow the user to select a file.

Because we hide the file input, we need a bit of JavaScript to trigger the file input when the user clicks on the preview image or the button next to it:

```

<th:block layout:fragment="page-scripts">
    <script>
        document.querySelector('#selectAvatarButton').addEventListener(
        'click', evt => { ①
            document.querySelector('#selectAvatarButton').blur();
            document.querySelector('#avatarFile').click();
        });
    </script>

```

```

        document.querySelector('#avatarImage').addEventListener('click',
evt => { ②
            document.querySelector('#avatarImage').blur();
            document.querySelector('#avatarFile').click();
        });

        document.querySelector('#avatarFile').addEventListener('change',
evt => { ③
            previewImage();
        });

        function previewImage() {
            var uploader = document.querySelector('#avatarFile');
            if (uploader.files && uploader.files[0]) {
                document.querySelector('#avatarImage').src = window.URL
.createObjectURL(uploader.files[0]); ④
                document.querySelector('#avatarImage').classList.remove
('p-6'); ⑤
            }
        }
    </script>
</th:block>

```

- ① Attach a `click` listener on the button so the file `<input>` is triggered when the button is clicked.
- ② Attach a `click` listener to the preview image so the file `<input>` is triggered when the button is clicked.
- ③ When the actual selected file changes, update the preview image
- ④ Set the `src` of the preview image to the uploaded file. Since we only allow to select a single file, we can safely use `uploader.files[0]`.
- ⑤ Remove the padding we need for the default SVG image when an actual avatar is now showing.

When we test all that in the browser, we should see this:

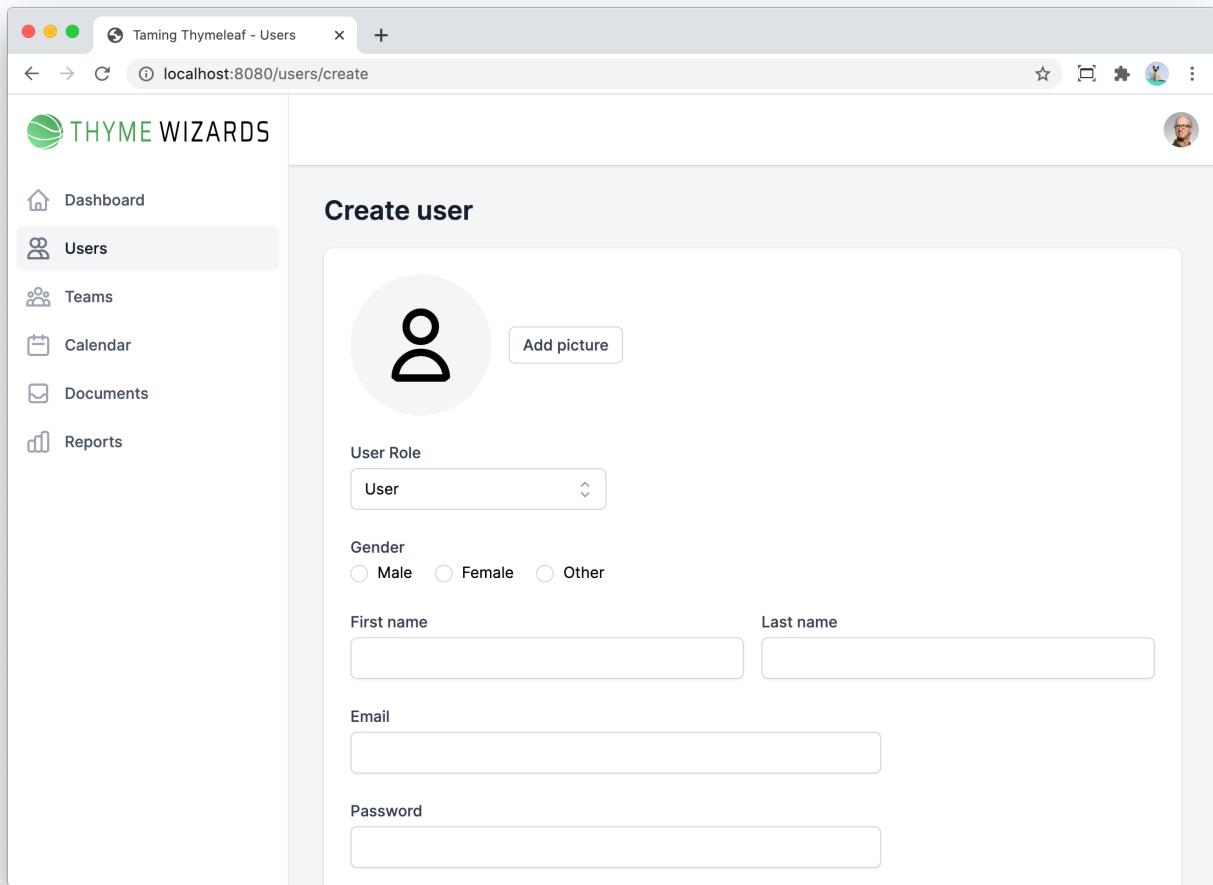


Figure 86. Form to select an avatar image for a new user

The user can select an avatar by either clicking on the dummy image, or by clicking on the 'Add picture' button.

When the user select an image, a preview is shown:

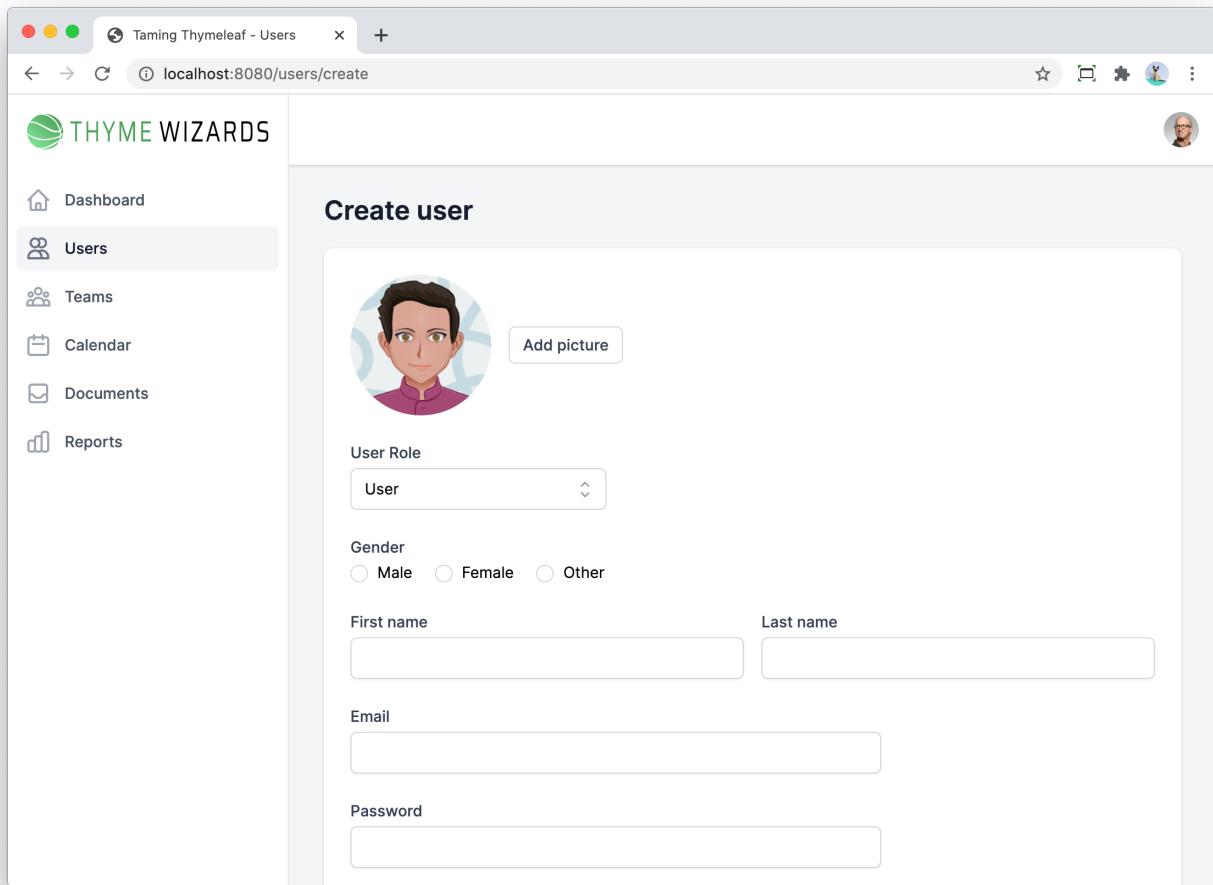


Figure 87. Preview of selected avatar file

When editing a user, we immediately see the preview if the user has an avatar:

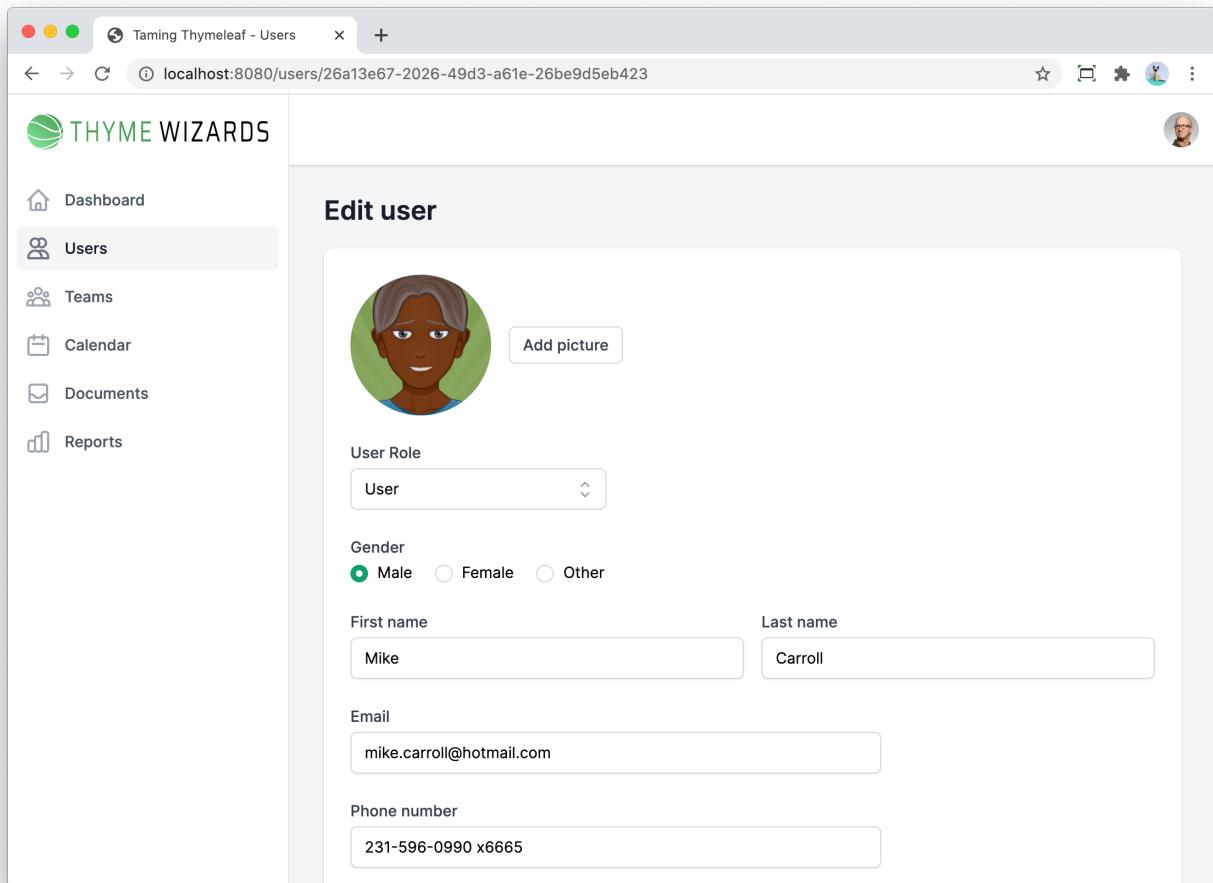


Figure 88. Preview of avatar when editing user

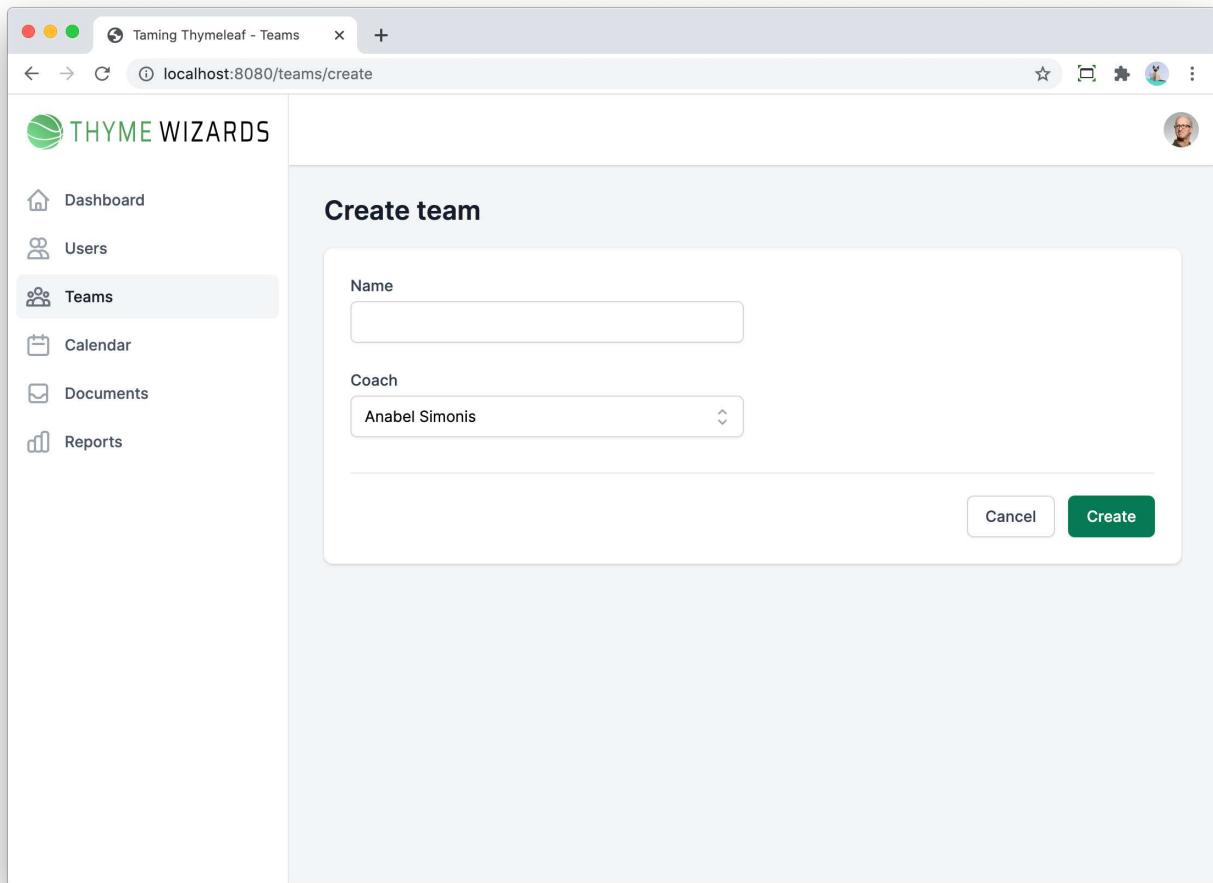
## 16.5. Selecting a linked entity value

### 16.5.1. Implementation

A common requirement in an application is selecting an entity from a list of entities to link that entity to another entity. It is probably a bit hard to understand that sentence, so let's make it practical as follows:

We will create a **Team** entity. Each team has a coach. When we create a form to create or edit a **Team**, we will have a combobox to select a coach. That combobox will contain all users of the application.

Creating a team will look like this:



We'll start by creating our Team entity using JPearl:

```
mvn jpearl:generate -Dentity=Team
```

Expanding on that generated code, we have our Team entity like this:

```
package com.tamingthymeleaf.application.team;

import com.tamingthymeleaf.application.user.User;
import io.github.wimdeblauwe.jpearl.AbstractVersionedEntity;

import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.ManyToOne;
import javax.validation.constraints.NotBlank;

@Entity
public class Team extends AbstractVersionedEntity<TeamId> {

    @NotBlank
    private String name;
```

```

@ManyToOne(fetch = FetchType.LAZY)
private User coach; ①

/**
 * Default constructor for JPA
 */
protected Team() {
}

public Team(TeamId id,
            String name,
            User coach) {
    super(id);
    this.name = name;
    this.coach = coach;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public User getCoach() {
    return coach;
}

public void setCoach(User coach) {
    this.coach = coach;
}
}

```

- ① Create a link between `Team` and `User` using the many-to-one relationship (A single coach might coach different teams)

To support this `Team` entity, we need to create a database table:

`src/main/resources/db/migration/V1.1_add-team.sql`

```

CREATE TABLE team
(
    id      UUID      NOT NULL,
    version BIGINT   NOT NULL,

```

```

name      VARCHAR NOT NULL,
coach_id  UUID     NOT NULL,
PRIMARY KEY (id)
);

ALTER TABLE team
ADD CONSTRAINT FK_team_to_user FOREIGN KEY (coach_id) REFERENCES
tt_user;

```

We will also create a [TeamService](#):

```

package com.tamingthymeleaf.application.team;

import com.tamingthymeleaf.application.user.User;
import com.tamingthymeleaf.application.user.UserId;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;

import java.util.Optional;

public interface TeamService {
    Page<TeamSummary> getTeams(Pageable pageable);

    Team createTeam(String name, User coach);

    Team createTeam(String name, UserId coachId);

    Optional<Team> getTeam(TeamId teamId);

    Team editTeam(TeamId teamId, long version, String name, UserId
coachId);

    void deleteTeam(TeamId teamId);

    void deleteAllTeams();
}

```



To keep things a bit simpler, we did *not* create a [CreateTeamParameters](#) object like we did for [CreateUserParameters](#). I would always create such an object for production code, unless there is only 1 or 2 properties needed to create the entity like we have here.

The [TeamServiceImpl](#) will use the [TeamRepository](#) for the database interaction:

```
package com.tamingthymeleaf.application.team;

import com.tamingthymeleaf.application.user.User;
import com.tamingthymeleaf.application.user.UserId;
import com.tamingthymeleaf.application.user.UserNotFoundException;
import com.tamingthymeleaf.application.user.UserService;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.orm.ObjectOptimisticLockingFailureException;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.Optional;

@Service
@Transactional
public class TeamServiceImpl implements TeamService {
    private static final Logger LOGGER = LoggerFactory.getLogger(TeamServiceImpl.class);
    private final TeamRepository repository;
    private final UserService userService;

    public TeamServiceImpl(TeamRepository repository, UserService userService) {
        this.repository = repository;
        this.userService = userService;
    }

    @Override
    @Transactional(readOnly = true)
    public Page<TeamSummary> getTeams(Pageable pageable) {
        return repository.findAllSummary(pageable);
    }

    @Override
    public Team createTeam(String name, User coach) {
        LOGGER.info("Creating team {} with coach {} ({})", name, coach
                .getUserName().getFullName(), coach.getId());
        return repository.save(new Team(repository.nextId(), name,
                coach));
    }
}
```

```

@Override
public Team createTeam(String name, UserId coachId) {
    User coach = getCoach(coachId);
    return createTeam(name, coach);
}

@Override
public Optional<Team> getTeam(TeamId teamId) {
    return repository.findById(teamId);
}

@Override
public Team editTeam(TeamId teamId, long version, String name,
UserId coachId) {
    Team team = getTeam(teamId)
        .orElseThrow(() -> new TeamNotFoundException(teamId));
    if (team.getVersion() != version) {
        throw new ObjectOptimisticLockingFailureException(User.
class, team.getId().asString());
    }

    team.setName(name);
    team.setCoach(getCoach(coachId));

    return team;
}

@Override
public void deleteTeam(TeamId teamId) {
    repository.deleteById(teamId);
}

@Override
public void deleteAllTeams() {
    repository.deleteAll();
}

private User getCoach(UserId coachId) {
    return userService.getUser(coachId)
        .orElseThrow(() -> new UserNotFoundException(
coachId));
}
}

```

The `TeamRepository` is a normal `CrudRepository`, but it uses a *Constructor Expression* to return a

Data Transfer Object (DTO). This allows us to only return the fields that we really need in an efficient query:

```
package com.tamingthymeleaf.application.team;

import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.CrudRepository;
import org.springframework.transaction.annotation.Transactional;

@Transactional(readOnly = true)
public interface TeamRepository extends CrudRepository<Team, TeamId>, TeamRepositoryCustom {
    @Query("SELECT new com.tamingthymeleaf.application.team.TeamSummary(t.id, t.name, t.coach.id, t.coach.userName) FROM Team t")
    Page<TeamSummary> findAllSummary(Pageable pageable);
}
```

This is the code for the `TeamSummary` DTO:

```
package com.tamingthymeleaf.application.team;

import com.tamingthymeleaf.application.user.UserId;
import com.tamingthymeleaf.application.user.UserName;

public class TeamSummary {
    private final TeamId id;
    private final String name;
    private final UserId coachId;
    private final UserName coachName;

    public TeamSummary(TeamId id, String name, UserId coachId, UserName coachName) {
        this.id = id;
        this.name = name;
        this.coachId = coachId;
        this.coachName = coachName;
    }

    public TeamId getId() {
        return id;
    }
}
```

```
public String getName() {
    return name;
}

public UserId getCoachId() {
    return coachId;
}

public UserName getCoachName() {
    return coachName;
}

}
```

For the create/edit form, we need a Java form data object to match. We will create [CreateTeamFormData](#) and [EditTeamFormData](#) for that purpose:

```
package com.tamingthymeleaf.application.team.web;

import com.tamingthymeleaf.application.user.UserId;

import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

public class CreateTeamFormData {
    @NotBlank
    @Size(max = 100)
    private String name;
    @NotNull
    private UserId coachId;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public UserId getCoachId() {
        return coachId;
    }
}
```

```

public void setCoachId(UserId coachId) {
    this.coachId = coachId;
}
}

```

Note how we are not adding a `User coach` field, but a `UserId coachId` field. The HTML `<select>` will match on the String representation of the `UserId`, not on the full Java object. For that reason, it is easier to implement this using `UserId`.

This is also why it is important to *not* just use your entity and map that to your form. It is much better to use dedicated form data objects like we do here.

The `EditTeamFormData` builds upon `CreateTeamFormData`:

```

package com.tamingthymeleaf.application.team.web;

import com.tamingthymeleaf.application.team.Team;

public class EditTeamFormData extends CreateTeamFormData {
    private String id;
    private long version;

    public static EditTeamFormData fromTeam(Team team) {
        EditTeamFormData result = new EditTeamFormData();
        result.setId(team.getId().asString());
        result.setVersion(team.getVersion());
        result.setName(team.getName());
        result.setCoachId(team.getCoach().getId());
        return result;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public long getVersion() {
        return version;
    }

    public void setVersion(long version) {
        this.version = version;
    }
}

```

```
}
```

```
}
```

Next, we create the `TeamController`, which is very similar to the `UserController` we created before:

```
package com.tamingthymeleaf.application.team.web;

import com.tamingthymeleaf.application.infrastructure.web.EditMode;
import com.tamingthymeleaf.application.team.Team;
import com.tamingthymeleaf.application.team.TeamId;
import com.tamingthymeleaf.application.team.TeamNotFoundException;
import com.tamingthymeleaf.application.team.TeamService;
import com.tamingthymeleaf.application.user.UserService;
import org.springframework.data.domain.Pageable;
import org.springframework.data.web.SortDefault;
import org.springframework.security.access.annotation.Secured;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.servlet.mvc.support.RedirectAttributes;

import javax.validation.Valid;

@Controller
@RequestMapping("/teams")
public class TeamController {

    private final TeamService service;
    private final UserService userService;

    public TeamController(TeamService service, UserService userService)
    {
        this.service = service;
        this.userService = userService;
    }

    @GetMapping
    public String index(Model model,
                        @SortDefault.SortDefaults(@SortDefault("name"))
Pageable pageable) {
        model.addAttribute("teams", service.getTeams(pageable));
        return "teams/list";
    }
}
```

```

}

@GetMapping("/create")
@Secured("ROLE_ADMIN")
public String createTeamForm(Model model) {
    model.addAttribute("team", new CreateTeamFormData());
    model.addAttribute("users", userService.getAllUsersNameAndId());
    return "teams/edit";
}

@PostMapping("/create")
@Secured("ROLE_ADMIN")
public String doCreateTeam(@Valid @ModelAttribute("team")
CreateTeamFormData formData,
                           BindingResult bindingResult, Model model)
{
    if (bindingResult.hasErrors()) {
        model.addAttribute("editMode",EditMode.CREATE);
        model.addAttribute("users", userService.
getAllUsersNameAndId());
        return "teams/edit";
    }

    service.createTeam(formData.getName(), formData.getCoachId());

    return "redirect:/teams";
}

@GetMapping("/{id}")
public String editTeamForm(@PathVariable("id") TeamId teamId,
                           Model model) {
    Team team = service.getTeam(teamId)
        .orElseThrow(() -> new TeamNotFoundException
(teamId));
    model.addAttribute("team", EditTeamFormData.fromTeam(team));
    model.addAttribute("users", userService.getAllUsersNameAndId());
    model.addAttribute("editMode",EditMode.UPDATE);
    return "teams/edit";
}

@PostMapping("/{id}")
@Secured("ROLE_ADMIN")
public String doEditTeam(@PathVariable("id") TeamId teamId,
                        @Valid @ModelAttribute("team")
EditTeamFormData formData,

```

```

        BindingResult bindingResult,
        Model model) {
    if (bindingResult.hasErrors()) {
        model.addAttribute("editMode", EditMode.UPDATE);
        model.addAttribute("users", userService.
getAllUsersNameAndId());
    }
    return "teams/edit";
}

service.editTeam(teamId, formData.getVersion(), formData.
getName(), formData.getCoachId());

return "redirect:/teams";
}

@PostMapping("/{id}/delete")
@Secured("ROLE_ADMIN")
public String doDeleteTeam(@PathVariable("id") TeamId teamId,
                           RedirectAttributes redirectAttributes) {
    Team team = service.getTeam(teamId)
        .orElseThrow(() -> new TeamNotFoundException
(teamId));

    service.deleteTeam(teamId);

    redirectAttributes.addFlashAttribute("deletedTeamName",
                                         team.getName());

    return "redirect:/teams";
}
}

```

What is important here is that we pass in the list of current users as `users` in the model. We don't pass the full `User` object as we only need the user name and id. For that purpose, a DTO was created:

```

package com.tamingthymeleaf.application.user;

public class UserNameAndId {
    private final UserId id;
    private final UserName userName;

    public UserNameAndId(UserId id, UserName userName) {
        this.id = id;
        this.userName = userName;
    }
}

```

```

    }

    public UserId getId() {
        return id;
    }

    public UserName getUserName() {
        return userName;
    }
}

```

This is used in the `UserService.getAllUsersNameAndId()` method:

`com.tamingthymeleaf.application.user.UserService`

```
ImmutableSortedSet<UserNameAndId> getAllUsersNameAndId();
```

With that Java code in place, we can create `src/main/resources/templates/teams/edit.html`:

```

<!DOCTYPE html>
<html
    xmlns:th="http://www.thymeleaf.org"
    xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
    layout:decorate="~{layout/layout}"
    th:with="activeMenuItem='teams'"

    <head>
        <title>Teams</title>
    </head>
    <body>
        <div layout:fragment="page-content">
            <div class="max-w-7xl mx-auto px-4 sm:px-6 md:px-8">
                <h1 class="text-2xl font-semibold text-gray-900"
                    th:text="${editMode?.name() == 'UPDATE'}?#{team.edit}:#{team.create}">Create team</h1>
            </div>
            <div class="max-w-7xl mx-auto px-4 sm:px-6 md:px-8">
                <div class="py-4">
                    <div class="bg-white shadow px-4 py-5 sm:rounded-lg sm:p-6">

                        <form id="team-form"
                            th:object="${team}"
                            th:action="${editMode?.name() == 'UPDATE'}?@{/teams/{id}(id=${team.id})}:@{/teams/create}">

```

```

        method="post"
        enctype="multipart/form-data">
        <div>
            <div th:replace="fragments/forms :: fielderrors"></div>
            <div class="mt-6 grid grid-cols-1 gap-y-6 gap-x-4 sm:grid-cols-6">
                <input type="hidden" th:field="*{version}" th:if="${editMode?.name() == 'UPDATE'}">

                <div th:replace="fragments/forms :: textinput(${team.name}, 'name', 'sm:col-span-3')"></div>
                <div class="sm:col-span-3"></div>
                <div class="sm:col-span-3">
                    <label for="coachId" class="block text-sm font-medium text-gray-700" th:text="#{team.coach}">
                    </label>
                    <div class="mt-1 rounded-md shadow-sm">
                        <select th:field="*{coachId}" class="max-w-lg block focus:ring-green-500 focus:border-green-500 w-full shadow-sm sm:max-w-xs sm:text-sm border-gray-300 rounded-md">
                            <option th:each="user : ${users}" th:text="${user.userName.fullName}" th:value="${user.id.asString()}"/>
                        </select>
                    </div>
                </div>
            </div>
            <div class="mt-8 border-t border-gray-200 pt-5">
                <div class="flex justify-end">
                    <span class="inline-flex rounded-md shadow-sm">
                        <button type="button" class="bg-white py-2 px-4 border border-gray-300 rounded-md shadow-sm text-sm font-medium text-gray-700 hover:bg-gray-50 focus:outline-none focus:ring-2 focus:ring-offset-2 focus:ring-green-500">
                            <th:text="#{cancel}">
                                Cancel
                            </th:text>
                        </button>
                    </span>
                </div>
            </div>
        </div>
    </div>
</div>

```

```

        </span>
            <span class="ml-3 inline-flex rounded-md shadow-sm">
                <button id="submit-button"
                    type="submit"
                    class="ml-3 inline-flex justify-center py-2 px-4 border border-transparent shadow-sm text-sm font-medium rounded-md text-white bg-green-600 hover:bg-green-700 focus:outline-none focus:ring-2 focus:ring-offset-2 focus:ring-green-500"
                    th:text="${editMode?.name() == 'UPDATE'}?#{save}:#{create}">
                    Save
                </button>
            </span>
        </div>
    </div>
</div>
</body>
</html>

```

Let's zoom in to the actual combobox code:

```

<div class="sm:col-span-3">
    <label for="coachId" class="block text-sm font-medium text-gray-700"
        th:text="#{team.coach}">
    </label>
    <div class="mt-1 rounded-md shadow-sm">
        <select th:field="*{coachId}"
            class="max-w-lg block focus:ring-green-500 focus:border-green-500 w-full shadow-sm sm:max-w-xs sm:text-sm border-gray-300 rounded-md">
            <option th:each="user : ${users}"
                th:text="${user.userName.fullName}"
                th:value="${user.id.asString()}">
            </select>
        </div>
    </div>

```

Important points:

- The `<select>` has a `th:field` attribute that references to the `coachId` property of the `CreateTeamFormData` and `EditTeamFormData` objects.
- We create as many `<option>` subtags as there are `users`.
- Use `th:text` for the visible text that the user will see.
- Use `th:value` for the value associated with the option (The primary key of the user in our case)

Thymeleaf will set the `<option>` to `selected` automatically for the tag where the `value` matches with the current `coachId`.

If we look at the page source in the browser, it will look something like this:

```
<select class="max-w-lg block focus:ring-green-500 focus:border-green-500 w-full shadow-sm sm:max-w-xs sm:text-sm border-gray-300 rounded-md"
        id="coachId"
        name="coachId">
    ...
    <option value="381e104f-4fe9-45d1-aa00-1e7679fb1bc4">Donita Koepp
    </option>
    <option value="13a38612-1b2b-441b-9dc3-42b039cd9fa3">Drew Herzog
    </option>
    <option value="b584818a-2c57-457c-a502-49ce87ac34a5"
selected="selected">Earle Wehner
    </option>
    <option value="ada75a4d-4d25-4338-a145-aee90c1cb4c8">Ed Corkery
    </option>
    <option value="99bbf611-1ee6-40b3-8874-965cbcba93b1">Emmett Bailey
    </option>
    ...
</select>
```

Screenshot of editing an existing team:

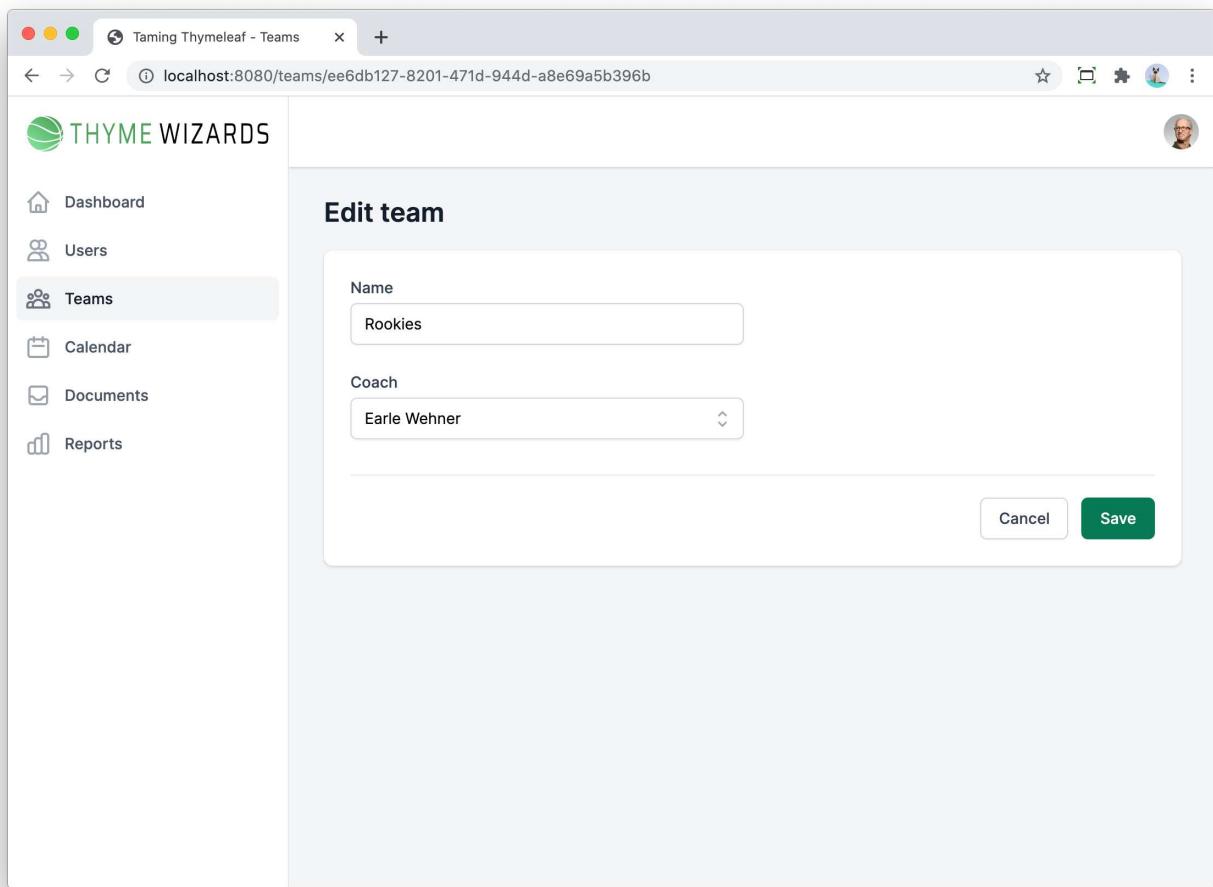


Figure 89. Editing the properties of a team

Screenshot of viewing the list of teams:

NAME	COACH		
Initiates	Myrle Oberbrunner	Edit	Delete
Knights	Karine Watsica	Edit	Delete
Padawans	Clementine Abernathy	Edit	Delete
Rookies	Earle Wehner	Edit	Delete
Wizards	Candra Steuber	Edit	Delete

Figure 90. List of the teams

### 16.5.2. Tests

To ensure everything works fine, we can add a few Cypress tests. Create `src/test/e2e/cypress/integration/team-management.spec.js` and add a test to add a team:

```
/// <reference types="Cypress" />

describe('Team management', () => {
  beforeEach(() => { ①
    cy.setCookie(
      'org.springframework.web.servlet.i18n.CookieLocaleResolver.LOCALE', 'en
    );
    cy.request({
      method: 'POST',
      url: 'api/integration-test/reset-db',
      followRedirect: false
    }).then(response) => {
      expect(response.status).to.eq(200);
    });
    cy.loginByForm('admin.strator@gmail.com', 'admin-pwd');
```

```

    cy.visit('/teams');
});

it('should be possible to add a team', () => { ②
  cy.get('#add-team-button').click();

  cy.url().should('include', '/teams/create');

  cy.get('#name').type('Wizards'); ③
  cy.get('#coachId').select('Admin Strator'); ④
  cy.get('#submit-button').click(); ⑤

  cy.get('#teams-table').find('tbody tr').should('have.length',
1); ⑥
};

});

```

- ① Reset the database and login before each test.
- ② Test to check if it is possible to add a new team.
- ③ Type a name for the team.
- ④ Select one of the users as coach for the team. Cypress allows to use the textual value (name of the user) or the id.
- ⑤ Submit the form.
- ⑥ Check if the new team is added in the list of teams.

We can also test if the delete functionality works fine. Let's first add a new endpoint to [IntegrationTestController](#) so there is a team present we can delete in the test:

```

@PostMapping("/reset-db")
public void resetDatabase() {
    teamService.deleteAllTeams(); ①
    userService.deleteAllUsers();

    addUser();
    addAdministrator();
}

@PostMapping("/add-test-team")
public void addTestTeam() {
    UserNameAndId userNameAndId = userService.getAllUsersNameAndId()
        .first(); ②
    teamService.createTeam("Test Team", userNameAndId.getId()); ③
}

```

- ① Delete all the teams when resetting the database. If we did not do this, PostgreSQL would give an exception due to foreign key constraints when we try to delete a user that is linked to a team.
- ② Get a random user to use as coach.
- ③ Create the test team.

Now we can write the Cypress test:

```
it('should be possible to delete a team', () => {
    cy.request({ ①
        method: 'POST',
        url: 'api/integration-test/add-test-team',
        followRedirect: false
    }).then((response) => {
        expect(response.status).to.eq(200);
    });

    // We should have 1 team to get started
    cy.visit('/teams'); ②
    cy.get('#teams-table').find('tbody tr').should('have.length',
1);

    cy.get('[id^=delete-link-]').click(); ③
    cy.get('#delete-modal-message').contains('Are you sure you want
to delete team Test Team?'); ④
    cy.get('#delete-modal-submit-button').click(); ⑤

    // There should be no team left after the delete
    cy.get('#teams-table').find('tbody tr').should('have.length',
0); ⑥

    cy.get('#success-alert-message').contains('Team Test Team was
deleted successfully.'); ⑦

    cy.reload(); ⑧
    cy.get('#success-alert-message').should('not.exist'); ⑨
});
```

- ① Call our new endpoint to create the test team.
- ② Visit the `/teams` URL so the webpage shows the team.
- ③ Find the delete link. Each delete link was given a unique id like `delete-link-<teamId>`. Using `get('[id^=delete-link-]')` allows us to match an element where the id starts with `delete-link-`. As we only have 1 team, there will be only 1 such link in this test.
- ④ Check if the modal message is showing the name of the team that is about to be deleted.
- ⑤ Confirm the delete.

- ⑥ Check that the list of teams is now empty.
- ⑦ Check that the delete confirmation message is shown.
- ⑧ Reload the page. Since we are using flash attributes for the confirmation message, the message should now be gone.
- ⑨ Validate that the message is no longer present after page reload.

While working on Cypress tests, it can be convenient to work on a single test only to avoid that all tests are run for each change you do. Replace `it` with `it.only` so that Cypress will run that test only:



```
it.only('should be possible to delete a team', () => {
    ...
})
```

## 16.6. Dynamically adding rows

This example will show how we can add and remove rows dynamically during editing without page refreshes. The example we build will allow an administrator to add users to a team and specify their position on the team.

The goal is to build something similar to this:

The diagram illustrates a user interface for managing player positions. It features a header labeled "Players". Below the header, there are two rows of controls. Each row contains a dropdown menu (e.g., "Andre Toy" and "Karin Yundt") followed by a position dropdown ("Point Guard" and "Center") and a "Remove" button with a trash can icon. At the bottom left is an "Add" button. At the bottom right are "Cancel" and "Save" buttons.

Each row allows to select a user and his position on the team. The 'Add' button below the rows allows to add an extra row. The 'Remove' link at each row allows to remove a single row, to remove a user from the team.

### 16.6.1. Entities

We will start by adding a `TeamPlayer` entity. This entity represents a player on a team at a certain position:

```
package com.tamingthymeleaf.application.team;

import com.tamingthymeleaf.application.user.User;
import io.github.wimdeblauwe.jpearl.AbstractEntity;

import javax.persistence.*;
import javax.validation.constraints.NotNull;

@Entity
public class TeamPlayer extends AbstractEntity<TeamPlayerId> {
    @ManyToOne(fetch = FetchType.LAZY)
    @NotNull
    private Team team; ①

    @OneToOne
    @NotNull
    private User player; ②

    @Enumerated(EnumType.STRING)
    @NotNull
    private PlayerPosition position; ③

    protected TeamPlayer() {
    }

    public TeamPlayer(TeamPlayerId id,
                      User player,
                      PlayerPosition position) {
        super(id);
        this.player = player;
        this.position = position;
    }

    public Team getTeam() {
        return team;
    }
}
```

```

public void setTeam(Team team) {
    this.team = team;
}

public User getPlayer() {
    return player;
}

public PlayerPosition getPosition() {
    return position;
}
}

```

- ① A `TeamPlayer` has a reference to the `Team` they belong to. This is a `@ManyToOne` relation since many players make up a team.
- ② The `player` field is the reference to the `User` object.
- ③ `PlayerPosition` is an `enum` that indicates what position this player will on the team.

The `PlayerPosition` is straightforward:

```

package com.tamingthymeleaf.application.team;

/**
 * See https://en.wikipedia.org/wiki/Basketball\_positions
 */
public enum PlayerPosition {
    POINT_GUARD,
    SHOOTING_GUARD,
    SMALL_FORWARD,
    POWER_FORWARD,
    CENTER
}

```

Note that we did not use `mvn jpearl:generate` here. This is because `TeamPlayer` exists only in the context of the `Team`. In Domain-Driven Design terminology, both `Team` and `TeamPlayer` are entities. Together, they form an `aggregate`. `Team` is the *aggregate root*.

It is common to only create a repository for the aggregate, not separate repositories for each entity in the aggregate. For that reason, we will not create a separate `TeamPlayerRepository`, but will expand the `TeamRepository`.

We need a way to create a primary key for a `TeamPlayer`, so we need to expand `TeamRepositoryCustom`:

```

package com.tamingthymeleaf.application.team;

```

```
public interface TeamRepositoryCustom {
    TeamId nextId();

    TeamPlayerId nextPlayerId(); ①
}
```

And the implementation in [TeamRepositoryImpl](#):

*com.tamingthymeleaf.application.team.TeamRepositoryImpl*

```
@Override
public TeamPlayerId nextPlayerId() {
    return new TeamPlayerId(generator.getNextUniqueId());
}
```

To store this in the database, we need to update our Flyway script:

*src/main/resources/db/migration/1.1\_add-team.sql*

```
CREATE TABLE team
(
    id        UUID      NOT NULL,
    version   BIGINT    NOT NULL,
    name      VARCHAR   NOT NULL,
    coach_id  UUID      NOT NULL,
    PRIMARY KEY (id)
);

ALTER TABLE team
    ADD CONSTRAINT FK_team_to_user FOREIGN KEY (coach_id) REFERENCES
tt_user;

CREATE TABLE team_player ①
(
    id        UUID      NOT NULL,
    team_id   UUID      NOT NULL, ②
    player_id UUID      NOT NULL, ③
    position   VARCHAR   NOT NULL, ④
    PRIMARY KEY (id)
);

ALTER TABLE team_player
    ADD CONSTRAINT FK_team_player_to_team FOREIGN KEY (team_id)
REFERENCES team; ⑤
ALTER TABLE team_player
```

```
ADD CONSTRAINT FK_team_player_to_user FOREIGN KEY (player_id)
REFERENCES tt_user;
```

- ① `team_player` database table
- ② Reference to the `Team`.
- ③ Reference to the `User`.
- ④ Storage of the `PlayerPosition` enum.
- ⑤ Foreign key constraints between the team player and the team, and the team player and the user.

Let's make sure this all works fine by adding a test on `TeamRepositoryTest`:

`com.tamingthymeleaf.application.team.TeamRepositoryTest`

```
@Test
void testSaveTeamWithPlayers() {
    User coach = userRepository.save(Users.createUser(new UserName
("Coach", "1")));
    User player1 = userRepository.save(Users.createUser(new
UserName("Player", "1")));
    User player2 = userRepository.save(Users.createUser(new
UserName("Player", "2")));
    User player3 = userRepository.save(Users.createUser(new
UserName("Player", "3")));

    TeamId id = repository.nextId();
    Team team = new Team(id, "Initiates", coach);
    team.addPlayer(new TeamPlayer(repository.nextPlayerId(),
player1, PlayerPosition.POINT_GUARD));
    team.addPlayer(new TeamPlayer(repository.nextPlayerId(),
player2, PlayerPosition.SHOOTING_GUARD));
    team.addPlayer(new TeamPlayer(repository.nextPlayerId(),
player3, PlayerPosition.CENTER));

    repository.save(team);

    entityManager.flush();
    entityManager.clear();

    assertThat(repository.findById(id))
        .hasValueSatisfying(team1 -> {
            assertThat(team1.getId()).isEqualTo(id);
            assertThat(team1.getCoach().getId()).isEqualTo(
coach.getId());
            assertThat(team1.getPlayers()).hasSize(3);
        });
}
```

```
}
```

### 16.6.2. Static server side rendering

The form that edits a `Team` has a backing form object `CreateTeamFormData` and `EditTeamFormData`. To allow adding/editing the users on that team, we will need to expand those objects.

We start by creating `TeamPlayerFormData` which contains the information of a single player and his position on the team:

```
package com.tamingthymeleaf.application.team.web;

import com.tamingthymeleaf.application.team.PlayerPosition;
import com.tamingthymeleaf.application.team.TeamPlayer;
import com.tamingthymeleaf.application.user.UserId;

import javax.validation.constraints.NotNull;

public class TeamPlayerFormData {
    @NotNull
    private UserId playerId;
    @NotNull
    private PlayerPosition position;

    public UserId getPlayerId() {
        return playerId;
    }

    public void setPlayerId(UserId playerId) {
        this.playerId = playerId;
    }

    public PlayerPosition getPosition() {
        return position;
    }

    public void setPosition(PlayerPosition position) {
        this.position = position;
    }

    public static TeamPlayerFormData fromTeamPlayer(TeamPlayer player) {
        TeamPlayerFormData result = new TeamPlayerFormData();
        result.setPlayerId(player.getPlayer().getId());
        result.setPosition(player.getPosition());
    }
}
```

```

        return result;
    }
}

```

We can now use this in `CreateTeamFormData` to model the information of the players in the HTML form:

```

package com.tamingthymeleaf.application.team.web;

import com.tamingthymeleaf.application.user.UserId;

import javax.validation.Valid;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

public class CreateTeamFormData {
    @NotBlank
    @Size(max = 100)
    private String name;
    @NotNull
    private UserId coachId;

    @NotNull
    @Size(min = 1)
    @Valid
    private TeamPlayerFormData[] players; ①

    public CreateTeamFormData() {
        this.players = new TeamPlayerFormData[]{new
TeamPlayerFormData()}; ②
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public UserId getCoachId() {
        return coachId;
    }
}

```

```

public void setCoachId(UserId coachId) {
    this.coachId = coachId;
}

public TeamPlayerFormData[] getPlayers() {
    return players;
}

public void setPlayers(TeamPlayerFormData[] players) {
    this.players = players;
}
}

```

- ① Use an array of `TeamPlayerFormData` objects to store the information that will be edited in the HTML form.

The `@NotNull` and `@Size(min = 1)` annotations ensure that there is always at least 1 player in a team.

The `@Valid` annotation ensures that the validation annotations on `TeamPlayerFormData` itself are also validated.

- ② We need to ensure the `players` property has a valid value because Spring MVC/Thymeleaf will bind to that.

With our form data updated, we can now turn our attention to `teams/edit.html`. To get started we iterate over all known players and output a fragment that allows editing the player information for each player:

`src/main/resources/templates/teams/edit.html`

```

<h3>Players</h3>
<div class="col-span-6 ml-4">
    <div id="teamplayer-forms"> ①
        <th:block th:each="player, iter : ${team.players}"> ②
            <div th:replace="teams/edit-teamplayer-fragment :: teamplayer-form(index=${iter.index})"></div> ③
        </th:block>
    </div>
    <div class="mt-4">
        <a href="#" ④
            class="py-2 px-4 border border-gray-300 rounded-md text-sm font-medium text-gray-700 hover:text-gray-500 focus:outline-none focus:border-blue-300 focus:shadow-outline-blue active:bg-gray-50 active:text-gray-800"
                id="add-extra-teamplayer-form-button"
                th:text="#{team.player.add.extra}"
        ></a> ④
    </div>
</div>

```

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

- ① The `teamplayer-forms` `<div>` contains all forms that edit the players on the team.
- ② We iterate over all existing players.
- ③ We use the fragment to edit a single player (see below) for each player
- ④ Button to add an extra player to the team. This does not do anything yet.

The `teams/edit.html` template uses a fragment `teamplayer-form` from the `teams/edit-teamplayer-fragment.html` file:

`src/main/resources/templates/teams/edit-teamplayer-fragment.html`

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org"
      lang="en">
<div th:fragment="teamplayer-form"
      class="col-span-6 flex items-stretch"
      th:id="${'teamplayer-form-section-' + __${index}__}"> ①
<div class="grid grid-cols-1 row-gap-6 col-gap-4 sm:grid-cols-6">
    <div class="sm:col-span-2">
        <div class="mt-1 rounded-md shadow-sm">
            <select class="max-w-lg block focus:ring-green-500
focus:border-green-500 w-full shadow-sm sm:max-w-xs sm:text-sm border-
gray-300 rounded-md"
                   th:field="*{players[__${index}__].playerId}"> ②
                <option th:each="user : ${users}"
                       th:text="${user.userName.fullName}"
                       th:value="${user.id.asString()}">
                </select>
            </div>
        </div>
        <div class="sm:col-span-2">
            <div class="mt-1 rounded-md shadow-sm">
                <select class="max-w-lg block focus:ring-green-500
focus:border-green-500 w-full shadow-sm sm:max-w-xs sm:text-sm border-
gray-300 rounded-md"
                   th:field="*{players[__${index}__].position}"> ③
                <option th:each="position : ${positions}"
                       th:text="#{'PlayerPosition.' + ${position}}"
                       th:value="${position}">
                </select>
            </div>
        </div>
        <div class="ml-1 sm:col-span-2 flex items-center text-green-600
hover:text-green-900">
```

```

<div class="h-6 w-6">
    <svg th:replace="trash"></svg>
</div>
<a href="#" class="ml-1"
    th:text="#{team.player.remove}"> ④
</a>
</div>
</div>
</html>

```

- ① Set the `id` to a unique name using the `index` parameter that should be passed to the template.
- ② Bind the `<select>` to the `playerId` property of the current player (using again the `index` parameter).
- ③ Bind the `<select>` to the `position` property of the current player (Both `playerId` and this `position` refer to the `TeamPlayerFormData` class on the Java side).
- ④ Add a remove button to remove the player again from the team. This does not do anything yet.

Add the necessary translations to `messages.properties`:

```

team.player.add.extra=Add
team.player.remove=Remove
PlayerPosition.POINT_GUARD=Point Guard
PlayerPosition.SHOOTING_GUARD=Shooting Guard
PlayerPosition.SMALL_FORWARD=Small Forward
PlayerPosition.POWER_FORWARD=Power Forward
PlayerPosition.CENTER=Center

```

Finally, we also need to update `TeamController` to add the list of possible `PlayerPosition` values:

```

@GetMapping("/create")
@Secured("ROLE_ADMIN")
public String createTeamForm(Model model) {
    model.addAttribute("team", new CreateTeamFormData());
    model.addAttribute("users", userService.getAllUsersNameAndId());
    model.addAttribute("positions", PlayerPosition.values()); ①
    return "teams/edit";
}

```

- ① Add all values of the `PlayerPosition` enum as a model attribute.

If we now test the code, the browser should look similar to this when adding a team:

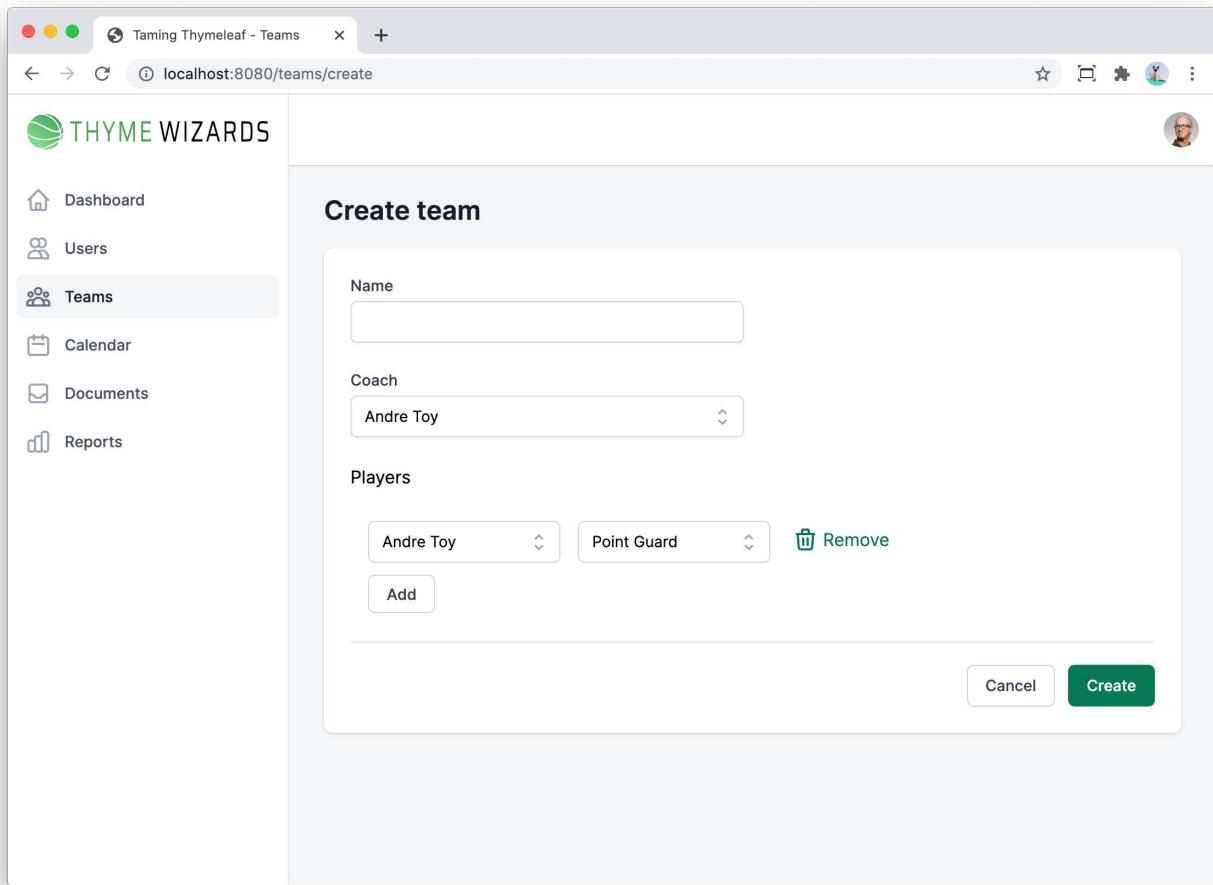


Figure 91. Creating a team with a single player

The code will also work for editing a team as this only needs the server-side Thymeleaf rendering we already implemented:

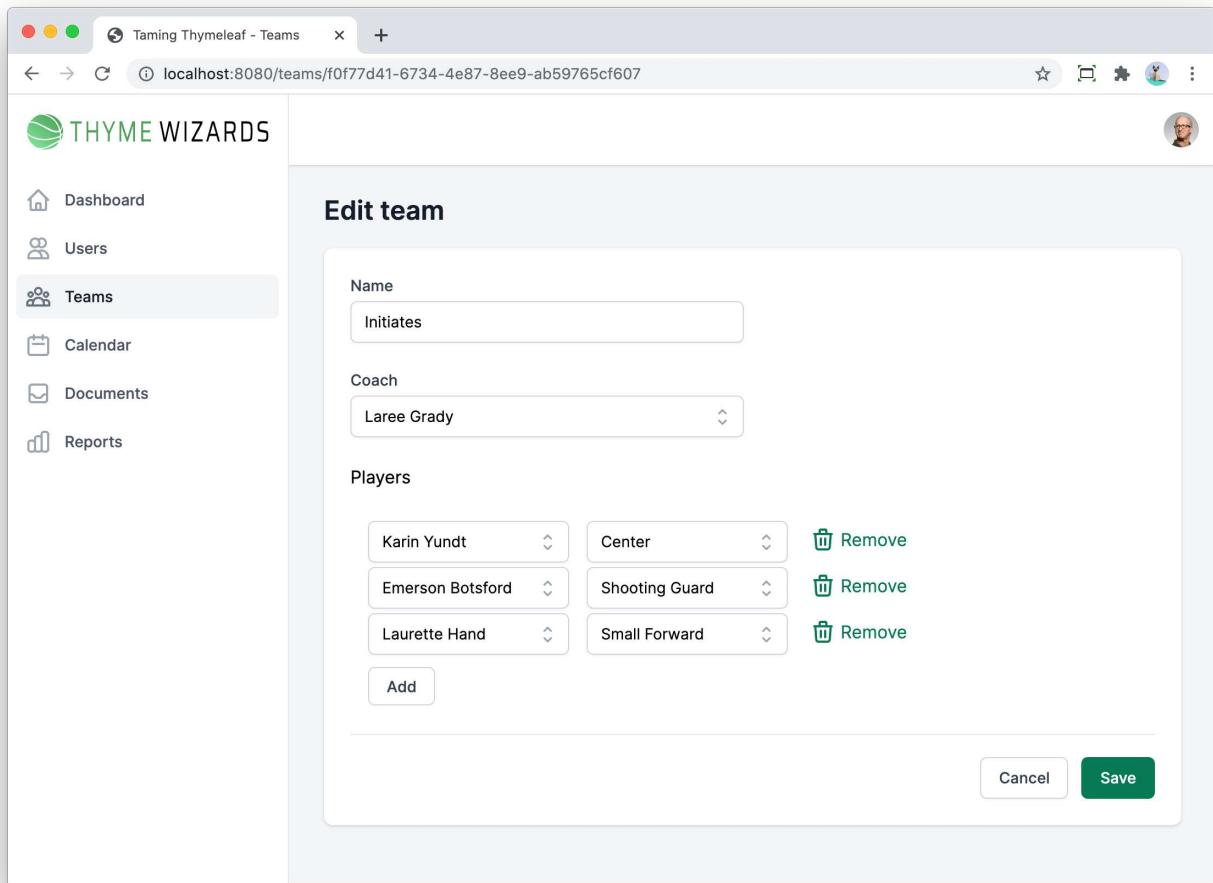


Figure 92. Editing a team with a few players

However, as we can't add players yet through the UI currently, we need to generate them when seeding the database.

`com.tamingthymeleaf.application.DatabaseInitializer`

```
Streams.forEachPair(generatedUsers.stream().limit(TEAM_NAMES.length),
    Arrays.stream(TEAM_NAMES),
    (user, teamName) -> {
        System.out.println(user);
        Team team = teamService.createTeam(teamName,
user);
        team = teamService.addPlayer(team.getId(), team
.getVersion(),
                randomUser(
generatedUsers), PlayerPosition.SMALL_FORWARD); ①
        team = teamService.addPlayer(team.getId(), team
.getVersion(),
                randomUser(
generatedUsers), PlayerPosition.SHOOTING_GUARD);
        teamService.addPlayer(team.getId(), team
.getVersion(),
```

```
        randomUser(
generatedUsers), PlayerPosition.CENTER);
});
```

- ① Add a random user as a `SMALL_FORWARD` to the team.

To make this code work, we need to update `TeamService` and `TeamServiceImpl`:

`com.tamingthymeleaf.application.team.TeamService`

```
Team addPlayer(TeamId id, long version, UserId userId,
PlayerPosition position);
```

`com.tamingthymeleaf.application.team.TeamServiceImpl`

```
@Override
public Team addPlayer(TeamId teamId, long version, UserId userId,
PlayerPosition position) {
    Team team = getTeam(teamId)
        .orElseThrow(() -> new TeamNotFoundException(teamId));
    if (team.getVersion() != version) {
        throw new ObjectOptimisticLockingFailureException(User.
class, team.getId().asString());
    }

    team.addPlayer(new TeamPlayer(repository.nextPlayerId(),
        getUser(userId),
        position));

    return team;
}
```

We also need to edit the `EditTeamFormData` to take the players into account when converting from the entity `Team` to the `EditTeamFormData` form backing object:

`com.tamingthymeleaf.application.team.web.EditTeamFormData`

```
public static EditTeamFormData fromTeam(Team team) {
    EditTeamFormData result = new EditTeamFormData();
    result.setId(team.getId().asString());
    result.setVersion(team.getVersion());
    result.setName(team.getName());
    result.setCoachId(team.getCoach().getId());
    result.setPlayers(team.getPlayers().stream()
        .map(TeamPlayerFormData::fromTeamPlayer)
        .toArray(TeamPlayerFormData[]::new)); ①
}
```

```
    return result;
}
```

- ① Convert each of the `TeamPlayer` objects to a `TeamPlayerFormData`.

The controller method for editing a team also needs an update:

`com.tamingthymeleaf.application.team.web.TeamController`

```
@GetMapping("/{id}")
public String editTeamForm(@PathVariable("id") TeamId teamId,
                           Model model) {
    Team team = service.getTeamWithPlayers(teamId) ①
        .orElseThrow(() -> new TeamNotFoundException
(teamId));
    model.addAttribute("team", EditTeamFormData.fromTeam(team));
    model.addAttribute("users", userService.getAllUsersNameAndId());
    model.addAttribute("positions", PlayerPosition.values()); ②
    model.addAttribute("editMode",EditMode.UPDATE);
    return "teams/edit";
}
```

- ① Because the `@OneToOne` mapping of `players` on the `Team` entity is lazy, we need to create a dedicated `TeamService` method that returns the Team with all the players.
- ② Add all `PlayerPosition` enum values.

Because we [disabled Open Session In View](#), we would get a `LazyInitializationException` inside `EditTeamFormData.fromTeam(team)` if we used `service.getTeam(teamId)`. This is because that method iterates over all the players, but there is no session anymore in the controller.

To avoid the problem, we add an extra method in `TeamRepository` to retrieve the team with the linked players:

```
package com.tamingthymeleaf.application.team;

import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.CrudRepository;
import org.springframework.data.repository.query.Param;
import org.springframework.transaction.annotation.Transactional;

import java.util.Optional;

@Transactional(readOnly = true)
public interface TeamRepository extends CrudRepository<Team, TeamId>,
TeamRepositoryCustom {
```

```

    @Query("SELECT new
com.tamingthymeleaf.application.team.TeamSummary(t.id, t.name,
t.coach.id, t.coach.userName) FROM Team t")
    Page<TeamSummary> findAllSummary(Pageable pageable);

    @Query("FROM Team t JOIN FETCH t.players WHERE t.id = :id")
    Optional<Team> findTeamWithPlayers(@Param("id") TeamId id); ①
}

```

① Use `JOIN FETCH` to retrieve the team with the linked players in a single SQL statement.

After this, we update `TeamService` and `TeamServiceImpl` to use this method.

Now run the application again with the `init-db` profile so the teams are generated with players in them. Editing a team should show the names of the players and their position in the team.

### 16.6.3. Make updates persistent

We can't add or remove rows yet, but we can edit the values of each player via the rows that are rendered by Thymeleaf on the server.

To make those edits persistent, we need to update `TeamController.doEditTeam()`. This currently looks like this:

```

    @PostMapping("/{id}")
    @Secured("ROLE_ADMIN")
    public String doEditTeam(@PathVariable("id") TeamId teamId,
                           @Valid @ModelAttribute("team")
    EditTeamFormData formData,
                           BindingResult bindingResult,
                           Model model) {
        if (bindingResult.hasErrors()) {
            model.addAttribute("editMode",EditMode.UPDATE);
            model.addAttribute("users", userService.
getAllUsersNameAndId());
            model.addAttribute("positions", PlayerPosition.values());
            return "teams/edit";
        }

        service.editTeam(teamId, formData.getVersion(), formData.
getName(), formData.getCoachId());

        return "redirect:/teams";
    }

```

We could expand the `editTeam` method with an extra parameter, but it will be better to use a

[Parameters](#) object.

Start with [TeamPlayerParameters](#):

```
package com.tamingthymeleaf.application.team;

import com.tamingthymeleaf.application.user.UserId;

import javax.validation.constraints.NotNull;

public class TeamPlayerParameters {
    private final UserId playerId;
    private final PlayerPosition position;

    public TeamPlayerParameters(UserId playerId, PlayerPosition position) {
        this.playerId = playerId;
        this.position = position;
    }

    public UserId getPlayerId() {
        return playerId;
    }

    public PlayerPosition getPosition() {
        return position;
    }
}
```

Using that, we can build [CreateTeamParameters](#):

```
package com.tamingthymeleaf.application.team;

import com.tamingthymeleaf.application.user.UserId;

import java.util.Set;

public class CreateTeamParameters {
    private final String name;
    private final UserId coachId;
    private final Set<TeamPlayerParameters> players;

    public CreateTeamParameters(String name, UserId coachId, Set<TeamPlayerParameters> players) {
```

```

        this.name = name;
        this.coachId = coachId;
        this.players = players;
    }

    public String getName() {
        return name;
    }

    public UserId getCoachId() {
        return coachId;
    }

    public Set<TeamPlayerParameters> getPlayers() {
        return players;
    }
}

```

And [EditTeamParameters](#):

```

package com.tamingthymeleaf.application.team;

import com.tamingthymeleaf.application.user.UserId;

import java.util.Set;

public class EditTeamParameters extends CreateTeamParameters {
    private final long version;

    public EditTeamParameters(long version, String name, UserId coachId,
Set<TeamPlayerParameters> players) {
        super(name, coachId, players);
        this.version = version;
    }

    public long getVersion() {
        return version;
    }
}

```

We can now update [TeamService](#) to use the parameter classes:

```

public interface TeamService {

```

```

Team createTeam(CreateTeamParameters parameters);

Team editTeam(TeamId teamId, EditTeamParameters parameters);

...
}

```

And also adjust the implementation in `TeamServiceImpl`:

```

public class TeamServiceImpl implements TeamService {
    @Override
    public Team createTeam(CreateTeamParameters parameters) {
        String name = parameters.getName();
        User coach = getUser(parameters.getCoachId());
        LOGGER.info("Creating team {} with coach {} ({})", name, coach
.getUser().getUserName(), coach.getFullName());
        Team team = new Team(repository.nextId(), name, coach);
        Set<TeamPlayerParameters> players = parameters.getPlayers();
        for (TeamPlayerParameters player : players) {
            team.addPlayer(new TeamPlayer(repository.nextPlayerId(),
getUser(player.getPlayerId()), player.getPosition()));
        }
        return repository.save(team);
    }

    @Override
    public Team editTeam(TeamId teamId, EditTeamParameters parameters) {
        Team team = getTeam(teamId)
            .orElseThrow(() -> new TeamNotFoundException(teamId));
        if (team.getVersion() != parameters.getVersion()) {
            throw new ObjectOptimisticLockingFailureException(User.
class, team.getId().asString());
        }

        team.setName(parameters.getName());
        team.setCoach(getUser(parameters.getCoachId()));
        team.setPlayers(parameters.getPlayers().stream()
            .map(teamPlayerParameters -> new
TeamPlayer(repository.nextPlayerId(), getUser(teamPlayerParameters
.getPlayerId()), teamPlayerParameters.getPosition()))
            .collect(Collectors.toSet()));

        return team;
    }
    ...
}

```

```
}
```

With our *domain* layer refactored, we can proceed to refactor the *web* layer. [CreateTeamFormData](#) now gets a method to convert to [CreateTeamParameters](#):

```
public class CreateTeamFormData {
    ...

    public CreateTeamParameters toParameters() {
        return new CreateTeamParameters(name, coachId,
getTeamPlayerParameters());
    }

    @Nonnull
    protected Set<TeamPlayerParameters> getTeamPlayerParameters() {
        return Arrays.stream(players)
            .map(teamPlayerFormData -> new
TeamPlayerParameters(teamPlayerFormData.getPlayerId(),
teamPlayerFormData.getPosition()))
            .collect(Collectors.toSet());
    }
}
```

Same for [EditTeamFormData](#):

```
public class EditTeamFormData extends CreateTeamFormData{
    ...

    @Override
    public EditTeamParameters toParameters() {
        return new EditTeamParameters(version,
            getName(),
            getCoachId(),
            getTeamPlayerParameters());
    }
}
```

Finally, we adjust [TeamController](#) to use the changes:

*com.tamingthymeleaf.application.team.web.TeamController*

```
@PostMapping("/create")
@Secured("ROLE_ADMIN")
```

```

public String doCreateTeam(@Valid @ModelAttribute("team")
CreateTeamFormData formData,
                           BindingResult bindingResult, Model model)
{
    if (bindingResult.hasErrors()) {
        model.addAttribute("editMode", EditMode.CREATE);
        model.addAttribute("users", userService.
getAllUsersNameAndId());
        model.addAttribute("positions", PlayerPosition.values());
        return "teams/edit";
    }

    service.createTeam(formData.toParameters());

    return "redirect:/teams";
}

@PostMapping("/{id}")
@Secured("ROLE_ADMIN")
public String doEditTeam(@PathVariable("id") TeamId teamId,
                        @Valid @ModelAttribute("team")
EditTeamFormData formData,
                           BindingResult bindingResult,
                           Model model) {
    if (bindingResult.hasErrors()) {
        model.addAttribute("editMode", EditMode.UPDATE);
        model.addAttribute("users", userService.
getAllUsersNameAndId());
        model.addAttribute("positions", PlayerPosition.values());
        return "teams/edit";
    }

    service.editTeam(teamId,
                     formData.toParameters());

    return "redirect:/teams";
}

```

Run the application again. You should be able to edit the user at a position in a team, or edit the position of a user.

#### 16.6.4. Add rows

This section will explain how to dynamically add new rows using JavaScript so no page refreshes are needed. Ideally, we will want to re-use the fragment we already used for generating the page. This will ensure the layout and functionality of the rows that are generated at page render are exactly the

same as the rows we add dynamically.

To make it possible to re-use the fragment, we have to slightly alter it:

```
<div th:fragment="teamplayer-form"
    class="col-span-6 flex items-stretch"
    th:id="${'teamplayer-form-section-' + __${index}__}"
    th:object="${__${teamObjectName}__}"> ①
```

① Extra `th:object` attribute.

We need to add an extra parameter `teamObjectName` that will tell the fragment the name of the `TeamFormData` binding object in the `Model`. When rendering "normally", this will be the name of the `th:object` we already have on the full form in `teams/edit.html`. When Thymeleaf needs to render the fragment alone, we will need to pass in a dummy object there, otherwise, Thymeleaf will not be able to render the fragment as there is no `th:object` to refer to for statements like `th:field="*{players[${index}].playerId}`

We need to update `teams/edit.html` to use that extra parameter in the fragment. While we are changing that, we will also do 2 other changes:

`src/main/resources/templates/teams/edit.html`

```
<h3>Players</h3>
<div class="col-span-6 ml-4">
    <div id="teamplayer-forms"
        th:data-teamplayers-count="${team.players.length}"> ①
        <th:block th:each="player, iter : ${team.players}">
            <div th:replace="teams/edit-teamplayer-fragment ::"
teamplayer-form(index=${iter.index}, teamObjectName='team')></div> ②
        </th:block>
    </div>
    <div class="mt-4">
        <a href="#" class="py-2 px-4 border border-gray-300 rounded-md text-sm font-medium text-gray-700 hover:text-gray-500 focus:outline-none focus:border-blue-300 focus:shadow-outline-blue active:bg-gray-50 active:text-gray-800"
            id="add-extra-teamplayer-form-button"
            th:text="#{team.player.add.extra}"
            @click="addExtraTeamPlayerForm()"
        ></a> ③
    </div>
</div>
```

① Add an attribute `data-teamplayers-count` that will help the JavaScript code to know what index to use next.

- ② Use the `teamObjectName` parameter.
- ③ Add an `@click` attribute to trigger the `addExtraTeamPlayerForm()` JavaScript method (see below) when the link is clicked.

Add a method to `TeamController` that returns the fragment. Note how we can use the same `::` syntax as we do when referencing a Thymeleaf fragment normally from another template:

`com.tamingthymeleaf.application.team.web.TeamController`

```

@GetMapping("/edit-teamplayer-fragment")
@Secured("ROLE_ADMIN")
public String getEditTeamPlayerFragment(Model model,
                                         @RequestParam("index") int
                                         index) { ①
    model.addAttribute("index", index); ②
    model.addAttribute("users", userService.getAllUsersNameAndId());
    ③
    model.addAttribute("positions", PlayerPosition.values()); ④
    model.addAttribute("teamObjectName", "dummyTeam"); ⑤
    model.addAttribute("dummyTeam", new
    DummyTeamForTeamPlayerFragment()); ⑥
    return "teams/edit-teamplayer-fragment :: teamplayer-form"; ⑦
}

private static class DummyTeamForTeamPlayerFragment {
    private TeamPlayerFormData[] players;

    public TeamPlayerFormData[] getPlayers() {
        return players;
    }

    public void setPlayers(TeamPlayerFormData[] players) {
        this.players = players;
    }
}

```

- ① The fragment has an `index` parameter that we need to fill in. We will receive this from the JavaScript AJAX call as a query parameter.
- ② Set the value of the query parameter as a model attribute. This will allow Thymeleaf to use it when rendering the fragment.
- ③ Pass in the `users` since the fragment needs that to render the dropdown with all the user names.
- ④ Pass in `PlayerPosition` values.
- ⑤ Set the `teamObjectName` to `dummyTeam` (This can be really any value you want, as long as it matches with the next line)
- ⑥ Add our `DummyTeamForTeamPlayerFragment` instance so Thymeleaf can do its binding stuff. Note that this is only done to render the HTML. We will *not* actually bind on

[DummyTeamForTeamPlayerFragment](#). Since the returned HTML is added inside a form that is bound to the actual [TeamFormData](#), those dynamic rows will get bound to that object.

- ⑦ Return the path to the fragment.

The last piece of the puzzle is the the AJAX call implementation in JavaScript to add a new row dynamically in [teams/edit.html](#):

`src/main/resources/templates/teams/edit.html`

```
<th:block layout:fragment="page-scripts">
    <script>
        function addExtraTeamPlayerForm() { ①
            const teamPlayerForms = document.getElementById('teamplayer-
forms'); ②
            const count = teamPlayerForms.getAttribute('data-
teamplayers-count'); ③
            fetch(`/teams/edit-teamplayer-fragment?index=${count}`) ④
                .then(response => response.text()) ⑤
                .then(fragment => {
                    teamPlayerForms.appendChild(htmlToElement
(fragment)); ⑥
                    teamPlayerForms.setAttribute('data-teamplayers-
count', parseInt(count) + 1); ⑦
                });
        }

        function htmlToElement(html) {
            const template = document.createElement('template');
            html = html.trim(); // Never return a text node of
whitespace as the result
            template.innerHTML = html;
            return template.content.firstChild;
        }
    </script>
</th:block>
```

- ① Declare a function to add an extra row. This will be bound to the 'Add' button using `@click="addExtraTeamPlayerForm()`
- ② Get the `<div>` with the `teamplayer-forms` id since we will add a new row there.
- ③ Get the current count which is added as an attribute when Thymeleaf renders the page.
- ④ Do the AJAX call using the [Fetch API](#). Pass in the current `count` as the `index` query parameter.
- ⑤ Get the text of the response.
- ⑥ Create a HTML [Node](#) from the HTML text so we can append it on the HTML page.
- ⑦ Increment the `count` and update the `data-teamplayers-count` attribute.

Pressing the 'Add' button now gives us a new row each time:

The screenshot shows a web application window titled 'Taming Thymeleaf - Teams'. The left sidebar has links for Dashboard, Users, Teams (which is selected), Calendar, Documents, and Reports. The main content area is titled 'Edit team'. It has fields for 'Name' (set to 'Initiates') and 'Coach' (set to 'Laree Grady'). Below these, under 'Players', there is a table-like structure with two columns: 'Player Name' and 'Position'. There are five rows of data: Emerson Botsford (Shooting Guard), Sondra Heathcote (Small Forward), Karin Yundt (Center), Alton Farrell (Point Guard), and Andre Toy (Point Guard). To the right of each position is a 'Remove' link. At the bottom of the 'Players' section is an 'Add' button. At the very bottom of the form are 'Cancel' and 'Save' buttons.

Figure 93. New row added without page refresh

### 16.6.5. Delete rows

All that is left now is to make the 'Remove' links work. We will add a bit of JavaScript again to remove the row that the 'Remove' button is part of.

Update the `edit-teamplayer-fragment`:

`src/main/resources/templates/teams/edit-teamplayer-fragment.html`

```
<div class="ml-1 sm:col-span-2 flex items-center text-green-600
hover:text-green-900">
    <div class="h-6 w-6">
        <svg th:replace="trash"></svg>
    </div>
    <a href="#" class="ml-1"
        th:text="#{team.player.remove}"
        x-data
        th:attr="data-formindex=__${index}__"
        @click="removeTeamPlayerForm($el.dataset.formindex)"> ①
    </a>
</div>
```

```
</a>
</div>
```

① We add 3 attributes to the `<a>` element:

- `x-data`: to define a new AlpineJS scope.
- `th:attr`: to add the current `index` parameter of the fragment so we can read this from JavaScript and know what row we should delete.
- `@click`: trigger the actual removal of the row when the link is clicked. `$el.dataset.formIndex` refers to the `data-formIndex` attribute of this `<a>` tag that we add using `th:attr`.

In `teams/edit.html`, we can add the JavaScript for the `removeTeamPlayerForm` method:

`src/main/resources/templates/teams/edit.html`

```
function removeTeamPlayerForm(formIndex) {
    const teamplayerForm = document.getElementById('teamplayer-form-section-' + formIndex);
    teamplayerForm.parentElement.removeChild(teamplayerForm);
}
```

Try this out, you should see that rows can now be removed as well.

There is however one situation that is not working properly and that is removing a row "in the middle". If you add a few rows and then remove one of the earlier rows and try to save, you will get a validation error.

This is because the HTML looks something like this with 3 rows for example:

```
<div class="col-span-6 flex items-stretch" id="teamplayer-form-section-0">
    ...
    <select name="players[0].playerId" ...> ... </select>
</div>
<div class="col-span-6 flex items-stretch" id="teamplayer-form-section-1">
    ...
    <select name="players[1].playerId" ...> ... </select>
</div>
<div class="col-span-6 flex items-stretch" id="teamplayer-form-section-2">
    ...
    <select name="players[2].playerId" ...> ... </select>
</div>
```

When we remove the middle row, we get:

```

<div class="col-span-6 flex items-stretch" id="teamplayer-form-section-0">
    ...
        <select name="players[0].playerId" ...> ... </select>
</div>
<div class="col-span-6 flex items-stretch" id="teamplayer-form-section-2">
    ...
        <select name="players[2].playerId" ...> ... </select>
</div>

```

So there is no `players[1]` anymore.

When the form is bound back the `TeamFormData` object when saving, Spring will insert an empty `TeamPlayerFormData` object at index 1. Because the properties of `TeamPlayerFormData` are `@NotNull`, the validation fails.

We can avoid this by removing those empty `TeamPlayerFormData` objects *before* the actual validation runs.

Create a `org.springframework.validation.Validator` implementation that uses the `Decorator` design pattern:

`com.tamingthymeleaf.application.team.web.TeamController`

```

private static class RemoveUnusedTeamPlayersValidator implements Validator { ①
    private final Validator validator;

    private RemoveUnusedTeamPlayersValidator(Validator validator) { ②
        this.validator = validator;
    }

    @Override
    public boolean supports(@Nonnull Class<?> clazz) {
        return validator.supports(clazz);
    }

    @Override
    public void validate(@Nonnull Object target, @Nonnull Errors errors) {
        if (target instanceof CreateTeamFormData) { ③
            CreateTeamFormData formData = (CreateTeamFormData)
target;
            formData.removeEmptyTeamPlayerForms(); ④
        }
    }
}

```

```

        }

        validator.validate(target, errors);
    }
}

```

- ① Implement the `org.springframework.validation.Validator` interface so we can register the class as a validator.
- ② Pass the original `Validator` as a constructor argument, so we delegate to it after we removed the empty forms in the `validate` method.
- ③ Check if the object we are validating is our `CreateTeamFormData` (or the `EditTeamFormData` subclass).
- ④ Tell the form data object to remove the empty forms.

To have the framework use this inner class of `TeamController`, we need to register it to the `WebDataBinder`:

`com.tamingthymeleaf.application.team.web.TeamController`

```

@InitBinder
public void initBinder(WebDataBinder binder) {
    binder.setValidator(new RemoveUnusedTeamPlayersValidator(binder
        .getValidator()));
}

```

The code of the `removeEmptyTeamPlayerForms()` method:

`com.tamingthymeleaf.application.team.web.CreateTeamFormData`

```

public void removeEmptyTeamPlayerForms() {
    setPlayers(Arrays.stream(players)
        .filter(this::isNotEmptyTeamPlayerForm)
        .toArray(TeamPlayerFormData[]::new));
}

private boolean isNotEmptyTeamPlayerForm(TeamPlayerFormData
formData) {
    return formData != null
        && formData.getPlayerId() != null
        && formData.getPosition() != null;
}

```

Try it out, even deleting rows in between other rows should work fine now.

This concludes this section on dynamically adding and removing rows in a form using a bit of JavaScript to avoid page refreshes. I hope this has given you valuable insights on how this can be implemented if you need to do this on your own projects.

## 16.7. Custom editors and formatters

With Spring MVC and Thymeleaf, we convert from the HTML `<input>` values to Java objects. Using `String` on the `...FormData` objects makes this trivial.

However, there might be cases where you want to directly bind to a richer object. This can be done by implementing a custom property editor or a custom formatter.

### 16.7.1. Custom editor

We will use `PhoneNumber` as an example. In `AbstractUserFormData`, we currently have this:

```
@NotNull
@Pattern(regexp = "[0-9.\\-() x/+]+", groups = ValidationGroupOne
.class)
private String phoneNumber;
```

If we want to use the `PhoneNumber` class instead of `String`, we need to create a custom property editor:

```
package com.tamingthymeleaf.application.user;

import org.apache.commons.lang3.StringUtils;

import java.beans.PropertyEditorSupport;

public class PhoneNumberPropertyEditor extends PropertyEditorSupport {
    ①

    @Override
    public void setAsText(String text) throws IllegalArgumentException {
        ②
        if (StringUtils.isNotBlank(text)) {
            this.setValue(new PhoneNumber(text)); ③
        } else {
            this.setValue(null); ④
        }
    }

    @Override
    public String getAsText() { ⑤
        PhoneNumber value = (PhoneNumber) getValue(); ⑥
        return value != null ? value.asString() : ""; ⑦
    }
}
```

```
}
```

- ① Extend from `PropertyEditorSupport` (Which is a Java SDK class, not a Spring class)
- ② Override the `setAsText(String text)` method. This method must implement the conversion from `String` to the custom type (`PhoneNumber` in our example).
- ③ If the `text` is not blank, create a `PhoneNumber` instance and set it as the `value`.
- ④ If the `text` is blank, set the `value` to `null`.
- ⑤ Override the `getAsText()` method to implement the conversion from the custom type to `String`.
- ⑥ Get the `value` property and cast it to `PhoneNumber`. We are sure this will be a `PhoneNumber` instance given the implementation of `setAsText`.
- ⑦ Return either an empty `String` if the value is `null`, or return a properly formatted `String` representation of the phone number.

This editor is not active by default, you need register it on the controller where you want to use it:

`com.tamingthymeleaf.application.user.web.UserController`

```
@InitBinder
public void initBinder(WebDataBinder binder) {
    binder.registerCustomEditor(PhoneNumber.class, new
PhoneNumberPropertyEditor());
}
```



If you want to enable it for all controllers, add it to a `@ControllerAdvice` class like we did for `StringTrimmerEditor`.

We can now update `AbstractUserFormData` to use `PhoneNumber` instead of `String`:

`com.tamingthymeleaf.application.user.web.AbstractUserFormData`

```
@NotNull
private PhoneNumber phoneNumber;
```

If you compare this to what we had before, you will note that the `@NotBlank` and `@Pattern` annotations are now replaced with `@NotNull`. This is because `@NotBlank` and `@Pattern` only work for `String` values, not `PhoneNumber` values. To ensure the phone number is still a valid format, we have to move the validation logic inside `PhoneNumber`:

```
public class PhoneNumber {
    private static final Pattern VALIDATION_PATTERN = Pattern.compile
("[0-9.\\-() x/+]+");
    private String phoneNumber;

    protected PhoneNumber() {
```

```

    }

    public PhoneNumber(String phoneNumber) {
        Assert.hasText(phoneNumber, "phoneNumber cannot be blank");
        Assert.isTrue(VALIDATION_PATTERN.asPredicate().test(
phoneNumber), "phoneNumber does not have proper format");
        this.phoneNumber = phoneNumber;
    }

    ...
}

```

One might argue that it should have been there already in the first place, as it is important for a [value object](#) to protect its invariants.

Because of the change in validation annotations, we also need to change the messages to show to the user.

We had this before:

```

NotNull.user.phoneNumber=Please enter the phone number.
Pattern.user.phoneNumber=Please enter a valid phone number.

```

Which is now replaced with:

```

NotNull.user.phoneNumber=Please enter the phone number.
typeMismatch.user.phoneNumber=Please enter a valid phone number.

```

The [NotNull](#) is triggered when the input is empty because we set the [value](#) in the editor to [null](#) and we have the [@NotNull](#) annotation in [AbstractUserFormData](#).

The [typeMismatch](#) is triggered when the [Assert.isTrue\(\)](#) line in the constructor of [PhoneNumber](#) fails.

### 16.7.2. Custom formatter

As an alternative to using [java.beans.PropertyEditorSupport](#), we can use [org.springframework.format.Formatter](#):

```

package com.tamingthymeleaf.application.user;

import org.springframework.format.Formatter;

import javax.annotation.Nonnull;
import java.text.ParseException;

```

```

import java.util.Locale;

public class PhoneNumberFormatter implements Formatter<PhoneNumber> { ①
    @Nonnull
    @Override
    public PhoneNumber parse(@Nonnull String text, @Nonnull Locale
locale) throws ParseException {
        return new PhoneNumber(text); ②
    }

    @Nonnull
    @Override
    public String print(@Nonnull PhoneNumber object, @Nonnull Locale
locale) {
        return object.asString(); ③
    }
}

```

① Implement the `Formatter` interface, generically typed to our `PhoneNumber` class.

② Use the `parse` method to convert from the user string to a `PhoneNumber` object.

③ Use the `print` method to convert from `PhoneNumber` to `String`

Note that the Spring framework guarantees no `null` values are passed to the `Formatter`, so this simplifies the implementation a bit.

We don't configure formatters on a controller-base, but globally for the application via the `WebMvcConfigurer`:

```

package com.tamingthymeleaf.application.infrastructure.web;

import com.tamingthymeleaf.application.user.PhoneNumberFormatter;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.format.FormatterRegistry;
import org.springframework.web.servlet.LocaleResolver;
import
org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import
org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
import org.springframework.web.servlet.i18n.CookieLocaleResolver;
import org.springframework.web.servlet.i18n.LocaleChangeInterceptor;

@Configuration
public class WebMvcConfiguration implements WebMvcConfigurer {
    @Bean

```

```

public LocaleResolver localeResolver() {
    return new CookieLocaleResolver();
}

@Bean
public LocaleChangeInterceptor localeInterceptor() {
    LocaleChangeInterceptor localeInterceptor = new
    LocaleChangeInterceptor();
    localeInterceptor.setParamName("lang");
    return localeInterceptor;
}

@Override
public void addInterceptors(InterceptorRegistry registry) {
    registry.addInterceptor(localeInterceptor());
}

@Override
public void addFormatters(FormatterRegistry registry) { ①
    registry.addFormatter(new PhoneNumberFormatter()); ②
}
}

```

① Override the `addFormatters()` method.

② Add the custom formatter to the `FormatterRegistry`.

To test this, remove `PhoneNumberPropertyEditor` and the `@InitBinder` annotated method from `UserController` that uses it. The application itself will work exactly the same as it did with the property editor.

*PropertyEditor or Formatter?*

When should you use `PropertyEditor` and when `Formatter` is a question you might have after reading this chapter.

Roughly, you can state that `PropertyEditor` is best used if you need to register conversion for a single controller.

In most cases a `Formatter` is probably best as it is a bit simpler to implement and has the additional benefit of passing the `Locale` of the user.

You can read more about formatters in the [Spring Field Formatting](#) chapter of the Spring reference documentation.



## 16.8. Date picker

The administrator has to enter the birthday of a user using the [ISO-8601 format](#) of `YYYY-MM-DD`. This is not very user friendly since there is no date picker to select a date, and most likely, this is not the

date format the user normally uses.

There are many date picker components freely available. We will implement one of them here to show how it can be done using Thymeleaf.

### 16.8.1. Duet Date Picker

The one we picked is the [Duet Date Picker](#) because it looks good, has extensive functionality, supports accessibility and is small (~10kb).

We start by adding the JavaScript and CSS of the component to the `<head>` section of our `layout.html` base template:

`src/main/resources/templates/layout/layout.html`

```
<head>
    <meta charset="UTF-8">
    <title layout:title-pattern="$LAYOUT_TITLE - $CONTENT_TITLE">Taming
    Thymeleaf</title>
    <meta http-equiv="X-UA-Compatible" content="IE=edge"/>
    <meta name="viewport" content="width=device-width, initial-
    scale=1"/>

    <script type="module"
src="https://cdn.jsdelivr.net/npm/@duetds/date-
picker@1.0.1/dist/duet/duet.esm.js"></script>
    <script nomodule src="https://cdn.jsdelivr.net/npm/@duetds/date-
picker@1.0.1/dist/duet/duet.js"></script>
    <link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/@duetds/date-
picker@1.0.1/dist/duet/themes/default.css"/>
    <link rel="stylesheet" href="https://rsms.me/inter/inter.css">
    <link rel="stylesheet" th:href="@{/css/application.css}">
</head>
```

Next, we create a `dateinput` component in the `forms.html` fragments:

`src/main/resources/templates/fragments/forms.html`

```
<div th:fragment="dateinput(labelText, fieldName, cssClass)"
    th:class="${cssClass}">
    <label th:for="${fieldName}" class="block text-sm font-medium
    leading-5 text-gray-700"
        th:text="${labelText}">
        Text input label
    </label>
    <div class="mt-1 relative rounded-md shadow-sm">
        <duet-date-picker th:identifier="${fieldName}">
```

```

        th:value="*__${fieldName}__"
        th:name="${fieldName}"
        class="w-full sm:text-sm sm:leading-5"

th:classappend="#${fields.hasErrors('__${fieldName}__')?'error-
border':''}">
    </duet-date-picker>
    <div th:if="#${fields.hasErrors('__${fieldName}__')}>
        class="absolute inset-y-0 right-0 pr-14 flex items-center
pointer-events-none">
            <svg class="h-5 w-5 text-red-500" fill="currentColor"
viewBox="0 0 20 20">
                <path fill-rule="evenodd"
                    d="M18 10a8 8 0 11-16 0 8 8 0 0116 0zm-7 4a1 1 0
11-2 0 1 1 0 012 0zm-1-9a1 1 0 00-1 1v4a1 1 0 102 0V6a1 1 0 00-1-1z"
                    clip-rule="evenodd"/>
            </svg>
        </div>
    </div>
    <p th:if="#${fields.hasErrors('__${fieldName}__')}>
        th:text="#${strings.listJoin(#fields.errors('__${fieldName}__'),
', ')}">
        class="mt-2 text-sm text-red-600" th:id="__${fieldName}__+-_
error">Field validation error message(s).</p>
</div>

```

This fragment is heavily based upon the `textinput` fragment we already had with the following changes:

- Replace `<input>` with `<duet-date-picker>`
- Use `th:identifier` to set the `id` of the element (per the [documentation](#))
- Because the component is not a simple `<input>` but a collection of `<div>` tags with an `<input>` somewhere, we need to follow the documentation of the component and set the `name` and the `value` attributes on `<duet-date-picker>`. We do this by using:
  - `th:value="*__${fieldName}__"`
  - `th:name="${fieldName}"`
- Set an `error-border` CSS class when there is validation error so the component can be styled the same way as our other `<input>` elements.
- Change the right padding for the validation error icon from `pr-3` to `pr-14`. Otherwise, it would be below the icon to open the date picker.

Some screenshots of the date picker in action:

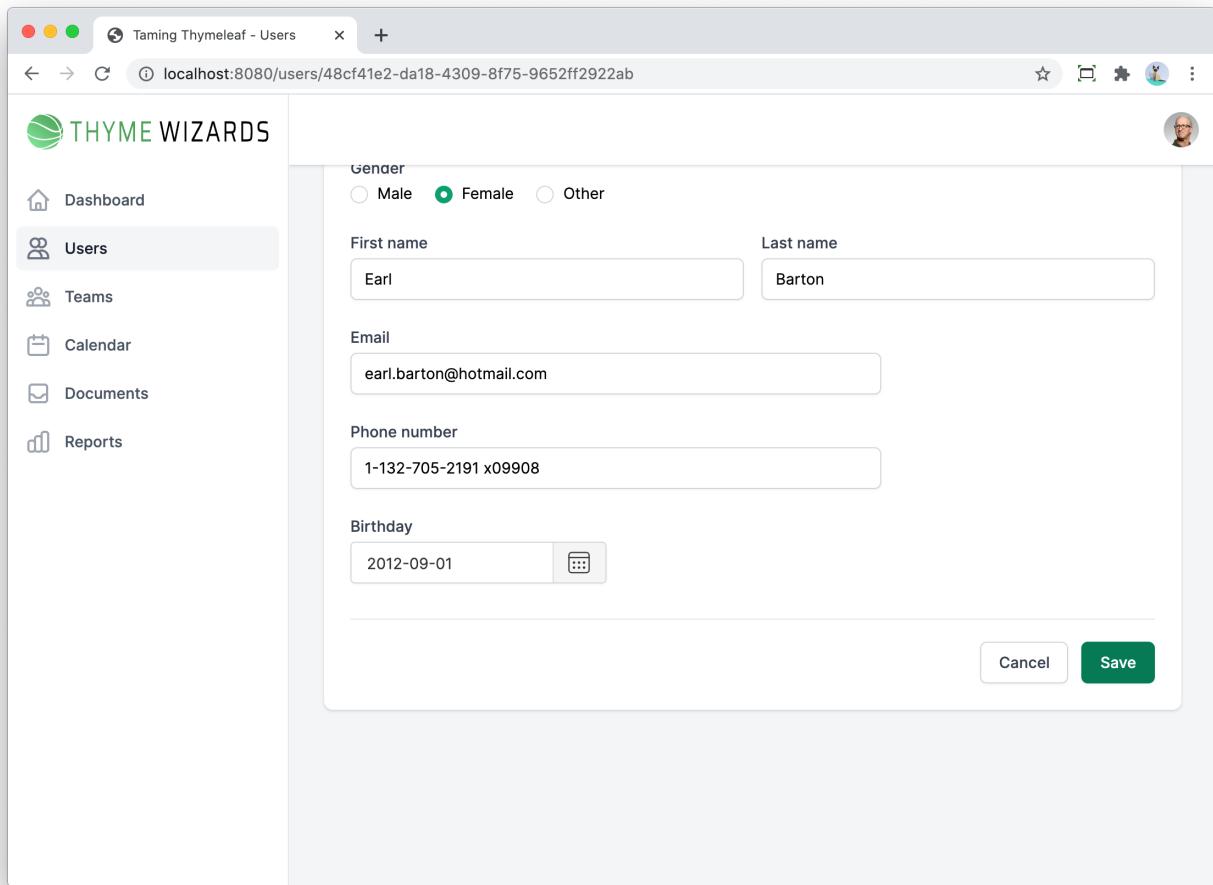


Figure 94. Birthday field using Duet Date Picker

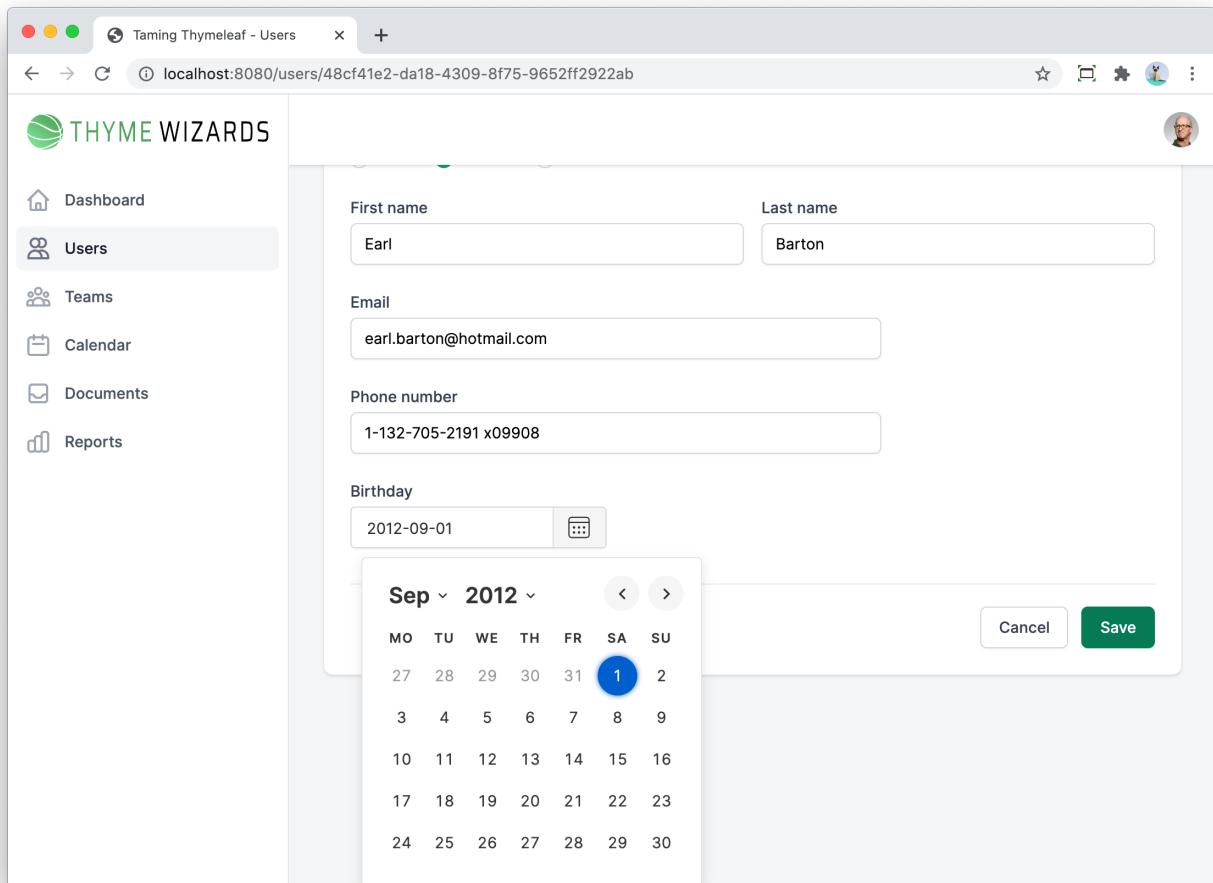


Figure 95. Date picker opened

The screenshot shows a web application interface for managing users. On the left, there's a sidebar with icons for Dashboard, Users (which is selected), Teams, Calendar, Documents, and Reports. The main area is titled "THYME WIZARDS". It contains fields for User Role (set to "User"), Gender (Female selected), First name ("Earl"), Last name ("Barton"), Email ("earl.barton@hotmail.com"), Phone number ("1-132-705-2191 x09908"), and Birthday (placeholder "YYYY-MM-DD" with an error icon). A message "Please enter the user's birthday." is displayed below the birthday field. At the bottom right are "Cancel" and "Save" buttons, with "Save" being green and highlighted.

Figure 96. No value selected shows error message and error icon

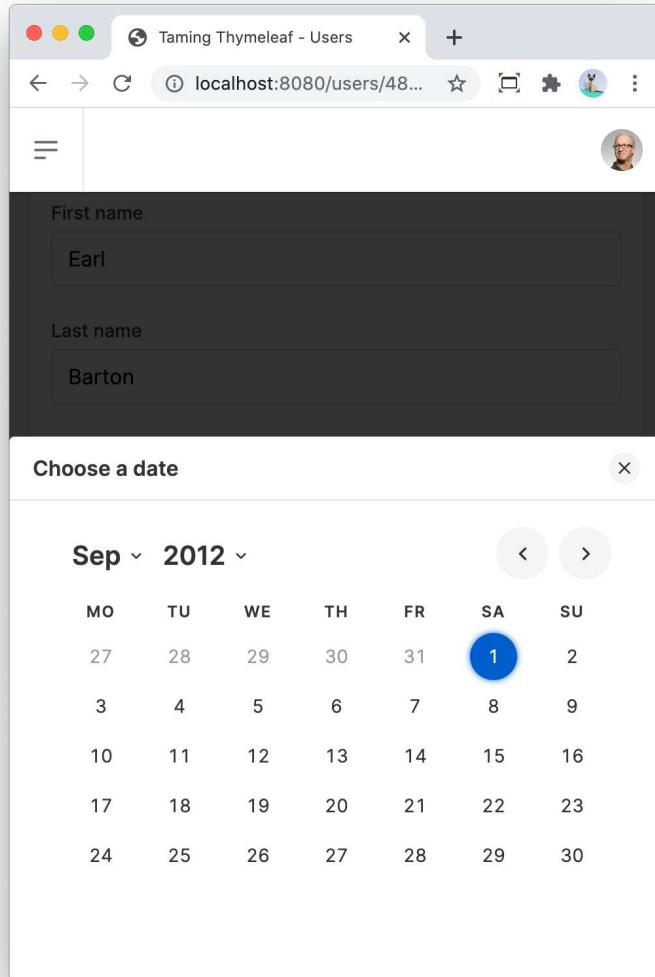


Figure 97. Date picker on mobile

### 16.8.2. Internationalization of date picker

The datepicker also has [localization support](#). We can have the input field show a textual, translated representation of the date by configurating the date picker using JavaScript.

To make all date pickers in the application consistent, we can add the JavaScript to [Layout.html](#).

First, add a dependency on the [Luxon](#) library to do the actual formatting:

`src/main/resources/templates/layout/layout.html`

```
<head>
  ...
  <script
    src="https://cdn.jsdelivr.net/npm/luxon@1.25.0/build/global/luxon.min.js"
  ></script>
  ...
</head>
```

Next, add the JavaScript:

`src/main/resources/templates/layout/layout.html`

```
<script>
    const picker = document.querySelector('duet-date-picker');

    if(picker) {
        picker.dateAdapter = {
            parse(value = '', createDate) { ①
                try {
                    let fromFormat = luxon.DateTime.fromFormat(value,
'yyyy-LL-dd');
                    if (fromFormat.isValid) {
                        return createDate(fromFormat.year, fromFormat
.month, fromFormat.day);
                    } else {
                        console.log('fromFormat not valid');
                    }
                } catch (e) {
                    console.log(e);
                }
            },
            format(date) { ②
                var DateTime = luxon.DateTime;
                return DateTime.fromJSDate(date) ③
                    .setLocale('[[${#strings.replace(#locale, '_', '_
-')}]])') ④
                    .toFormat('d LLLL yyyy'); ⑤
            },
        };
    }
</script>
```

- ① The `parse` function is used when a user manually types in the input field of the date picker to parse whatever the user is typing into a date. The code of this method allows to type in ISO-8601 format.
- ② The `format` function is the function that is called to format the date in the input field.
- ③ Create a Luxon `DateTime` object from the passed in JavaScript date object.
- ④ Pass the current locale to the Luxon object. Thymeleaf has the built-in `#locale` variable out-of-the-box. However, this is represented as `en_US` for example, while Luxon expects `en-US`. For that reason, we need to use the `#strings.replace()` function.
- ⑤ Specify the format of the `DateTime`.

We now have a nicely formatted date in our date picker:

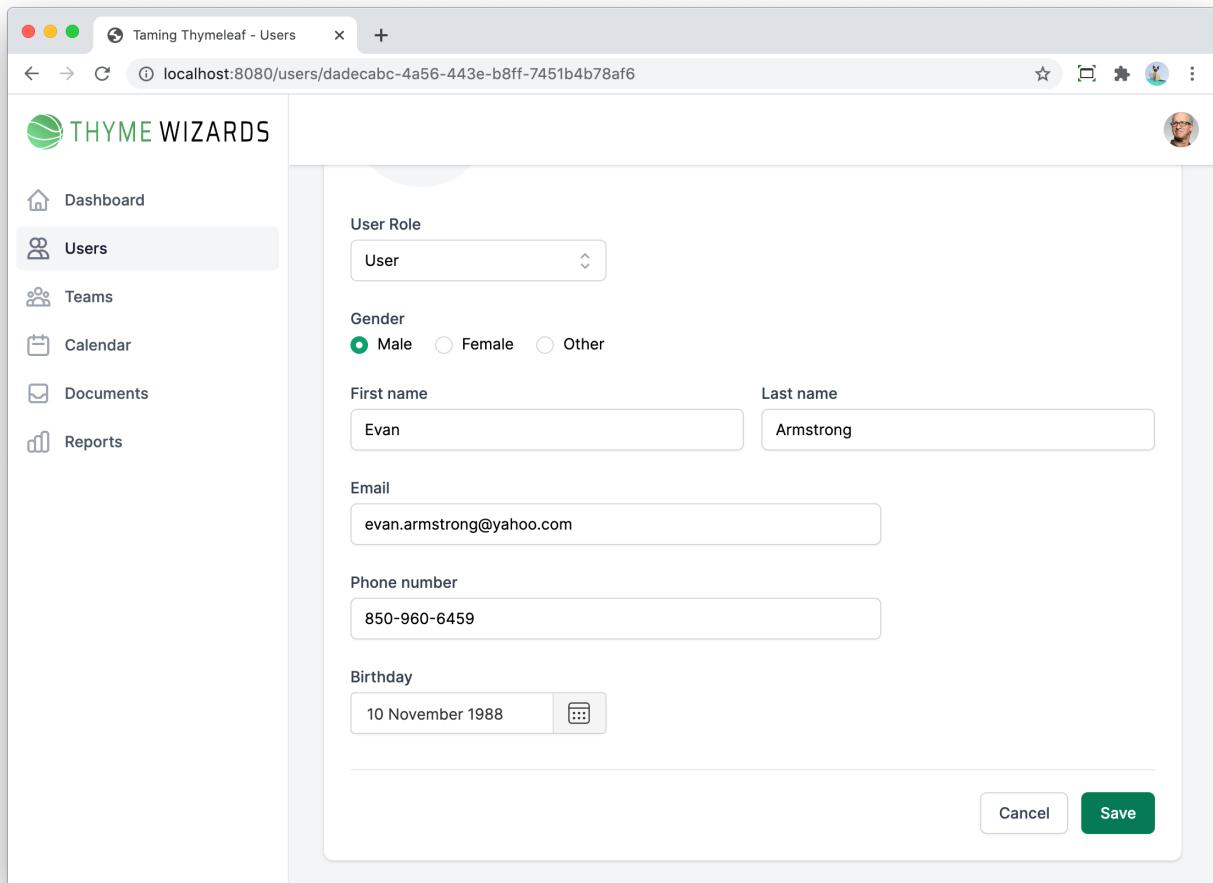


Figure 98. Date picker showing formatted date

# Chapter 17. Closing

We have come to the end of the Taming Thymeleaf book. I really hope you have enjoyed reading it and it has helped you grow as a developer.

Feel free to contact me at [wim.deblauwe@gmail.com](mailto:wim.deblauwe@gmail.com) or via [Twitter](#) if you have any remarks on the book.

If you want some further reading on Thymeleaf, I found these resources to be highly valuable:

- [The official Thymeleaf documentation](#), especially the appendix on [Expression Utility Objects](#) shows some very interesting helper functions that Thymeleaf has built-in.
- [Spring Web MVC documentation](#)
- My [personal blog](#) where I continue to write about [Thymeleaf-related topics](#).
- If you are stuck on a particular problem, [Stack Overflow](#) is the best place to ask your question. Be sure to tag it with the [Thymeleaf](#) tag.
- The [Cypress docs](#) if you want to know more details about writing tests with Cypress.
- The [Tailwind CSS](#) documentation is very nice if you want to learn about this utility-first CSS framework.

# Appendix A: Change log

## 2.0.2

*Oct 19, 2022*

- Fixed rendering of code snippet in chapter 10.

## 2.0.1

*Dec 31, 2021*

- Fixed typo in chapter [Link to URLs](#).
- Changed order of instructions in chapter 4. It is needed to run `npx tailwind init` first before running `npm run build` for the first time. Otherwise the command fails with an error.

## 2.0.0

*Dec 23, 2021*

- Update for Java 17, Spring Boot 2.6 and Tailwind CSS 3
- Also updated to Thymeleaf Layout Dialect 3.0.0, Alpine 3.7.0, Testcontainers 1.16.2, Guava 31.0.1 and Cypress 9.1.0

## 1.1.1

*Apr 13, 2021*

- Fix purging for production. To do this properly with Tailwind CSS 2, we need to use the `purge` option in `tailwind.config.js`. Updated chapter 4 to reflect this.

## 1.1.0

*Feb 16, 2021*

- All code samples updated for Tailwind 2

## 1.0.1

*Dec 26, 2020*

- Chapter 14.4.1: Add code to show that the table header also needs to be wrapped with `<th:block sec:authorize="hasRole('ADMIN')">`
- Chapter 15.1: Fix typo
- Chapter 15.1.3: Add note that tests require an `id` on the HTML elements so the tests can easily

find the elements.

- Chapter 2: Add example Maven version output for Windows (Reported by [Sai Upadhyayula](#))
- Add closing chapter

## 1.0.0

*Dec 5, 2020*

- Initial release