CHAPTER 3

Spring MVC

Spring Boot will automatically configure a web application when it finds the classes on the classpath. It will also start an embedded server (by default it will launch an embedded Tomcat)[1]

3-1 Getting Started with Spring MVC

Problem

You want to use Spring Boot to power a Spring MVC application.

Solution

Spring Boot will do auto-configuration for the components needed for Spring MVC. To enable this, Spring Boot needs to be able to detect the Spring MVC classes on its classpath. For this you will need to add the `spring-boot-starter-web` as a dependency.

How It Works

In your project, add the dependency for `spring-boot-starter-web`.

```xml
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

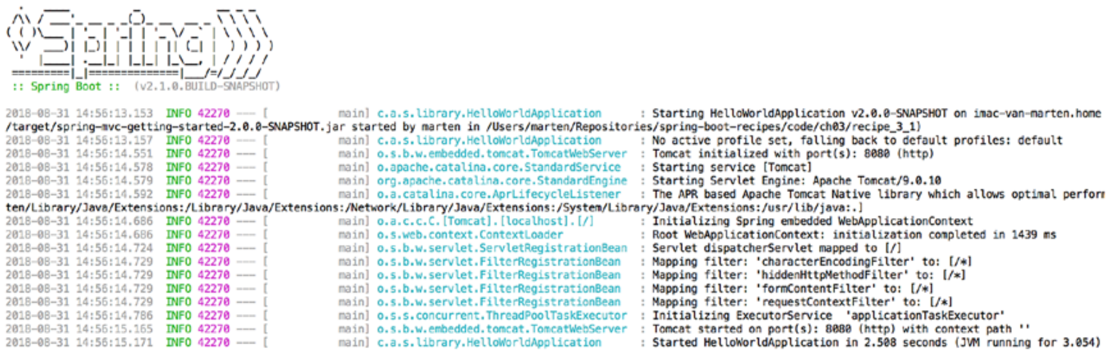
[1] https://tomcat.apache.org

© Marten Deinum 2018
M. Deinum, *Spring Boot 2 Recipes*, https://doi.org/10.1007/978-1-4842-3963-6_3

What this does is add the needed dependencies for Spring MVC. Now that Spring Boot can detect these classes, it will do additional configuration to set up the `DispatcherServlet`. It will also add all the JAR files needed to be able to start an embedded Tomcat server.

```java
package com.apress.springbootrecipes.hello;

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

@SpringBootApplication
public class HelloWorldApplication {

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

These few lines of code are enough to start the embedded Tomcat server and have a preconfigured Spring MVC setup. When you start the application, you will see output similar to that in Figure 3-1.



```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::  (v2.1.0.BUILD-SNAPSHOT)

2018-08-31 14:56:13.153  INFO 42270 --- [           main] c.a.s.library.HelloWorldApplication      : Starting HelloWorldApplication v2.0.0-SNAPSHOT on imac-van-marten.home
/target/spring-mvc-getting-started-2.0.0-SNAPSHOT.jar started by marten in /Users/marten/Repositories/spring-boot-recipes/code/ch03/recipe_3_1)
2018-08-31 14:56:13.157  INFO 42270 --- [           main] c.a.s.library.HelloWorldApplication      : No active profile set, falling back to default profiles: default
2018-08-31 14:56:14.551  INFO 42270 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
2018-08-31 14:56:14.578  INFO 42270 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2018-08-31 14:56:14.579  INFO 42270 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet Engine: Apache Tomcat/9.0.10
2018-08-31 14:56:14.592  INFO 42270 --- [           main] o.a.catalina.core.AprLifecycleListener   : The APR based Apache Tomcat Native library which allows optimal perfor
ten/Library/Java/Extensions:/Library/Java/Extensions:/Network/Library/Java/Extensions:/System/Library/Java/Extensions:/usr/lib/java:.]
2018-08-31 14:56:14.686  INFO 42270 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2018-08-31 14:56:14.686  INFO 42270 --- [           main] o.s.web.context.ContextLoader            : Root WebApplicationContext: initialization completed in 1439 ms
2018-08-31 14:56:14.724  INFO 42270 --- [           main] o.s.b.w.servlet.ServletRegistrationBean  : Servlet dispatcherServlet mapped to [/]
2018-08-31 14:56:14.729  INFO 42270 --- [           main] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'characterEncodingFilter' to: [/*]
2018-08-31 14:56:14.729  INFO 42270 --- [           main] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'hiddenHttpMethodFilter' to: [/*]
2018-08-31 14:56:14.729  INFO 42270 --- [           main] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'formContentFilter' to: [/*]
2018-08-31 14:56:14.729  INFO 42270 --- [           main] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'requestContextFilter' to: [/*]
2018-08-31 14:56:14.786  INFO 42270 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor  : Initializing ExecutorService  'applicationTaskExecutor'
2018-08-31 14:56:15.165  INFO 42270 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
2018-08-31 14:56:15.171  INFO 42270 --- [           main] c.a.s.library.HelloWorldApplication      : Started HelloWorldApplication in 2.508 seconds (JVM running for 3.054)
```

Figure 3-1. *Startup output logging*

The following things happen when you start the `HelloWorldApplication`:

1. Start an embedded Tomcat server on port 8080 (by default)
2. Register and enable a couple of default Servlet Filters (Table 3-1)

3. Set up static resource handling for things like `.css`, `.js` and `favicon.ico`
4. Enable integration with WebJars[2]
5. Setup basic error handling features
6. Preconfigure the `DispatcherServlet` with the needed components (i.e., `ViewResolvers`, I18N, etc.)

Table 3-1. *Automatically Registered Servlet Filters*

Filter	Description
`CharacterEncodingFilter`	Will force the encoding to be `UTF-8` by default, can be configured by setting the `spring.http.encoding.charset` property. Can be disabled through setting the `spring.http.encoding.enabled` to `false`
`HiddenHttpMethodFilter`	Enables the use of hidden form field named `_method` to specify the actual HTTP method. Can be disabled by setting the `spring.mvc.hiddenmethod.filter.enabled` to `false`
`FormContentFilter`	Will wrap the request for `PUT`, `PATCH`, and `DELETE` requests so that those can also benefit from binding. Can be disabled by setting the `spring.mvc.formcontent.filter.enabled` to `false`
`RequestContextFilter`	Exposes the current request to the current thread so that you can use the `RequestContextHolder` and `LocaleContextHolder` even in a non-Spring MVC application like Jersey

In the current state, the `HelloWorldApplication` doesn't do anything but start the server. Let's add a controller to return some information.

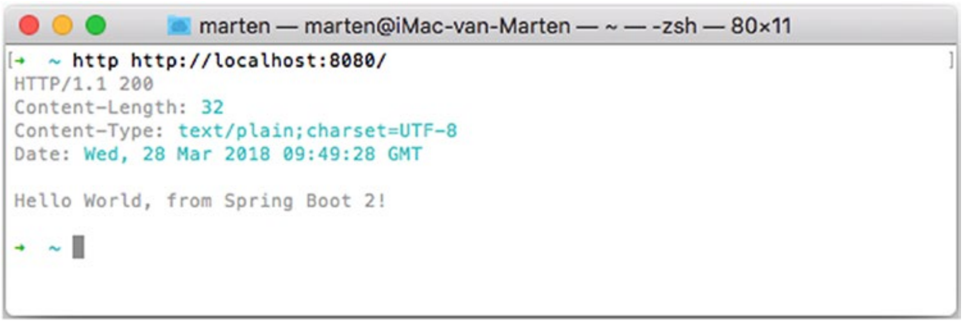
```
package com.apress.springbootrecipes.hello;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
```

[2]https://www.webjars.org

```java
@RestController
public class HelloWorldController {

  @GetMapping("/")
  public String hello() {
    return "Hello World, from Spring Boot 2!";
  }
}
```

This `HelloWorldController` will be registered at the `/` URL and when called will return the phrase `Hello World from Spring Boot 2!`. The `@RestController` indicates that this is a `@Controller` and as such will be detect by Spring Boot. Additionally, it adds the `@ResponseBody` annotation to all request handling methods, indicating it should send the result to the client. The `@GetMapping` maps the `hello` method to every GET request that arrives at `/`. We could also have written `@RequestMapping(value="/", method=RequestMethod.GET)`.

When restarting the `HelloWorldApplication`, the `HelloWorldController` will be detected and processed. Now when using something like `curl` or `http` to access `http://localhost:8080/` the result should be like that in Figure 3-2.



```
marten — marten@iMac-van-Marten — ~ — -zsh — 80×11
→  ~ http http://localhost:8080/
HTTP/1.1 200
Content-Length: 32
Content-Type: text/plain;charset=UTF-8
Date: Wed, 28 Mar 2018 09:49:28 GMT

Hello World, from Spring Boot 2!

→  ~ 
```

Figure 3-2. *Output of controller*

Testing

Now that the application is running and returning results, it's time to add a test for the controller (ideally you would write the test first!). Spring already has some impressive testing features and Spring Boot adds more of those. Testing a controller has become pretty easy with Spring Boot.

```
package com.apress.springbootrecipes.hello;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.request.MockMvcRequestBuilders;

import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

@RunWith(SpringRunner.class)
@WebMvcTest(HelloWorldController.class)
public class HelloWorldControllerTest {

  @Autowired
  private MockMvc mockMvc;

  @Test
  public void testHelloWorldController() throws Exception {
    mockMvc.perform(MockMvcRequestBuilders.get("/"))
      .andExpect(status().isOk())
      .andExpect(content().string("Hello World, from Spring Boot 2!"))
      .andExpect(content().contentTypeCompatibleWith(MediaType.TEXT_PLAIN));
  }
}
```

The `@RunWith(SpringRunner.class)` is needed to instruct JUnit to use this specific runner. This special runner is what integrates the Spring Test framework with JUnit. The `@WebMvcTest` instructs the Spring Test framework to set up an application context for testing this specific controller. It will start a minimal Spring Boot application with only the web-related beans like `@Controller`, `@ControllerAdvice`, etc. In addition it will preconfigure the Spring Test Mock MVC support, which can then be autowired.

Spring Test Mock MVC can be used to simulate making an HTTP request to the controller and do some expectations on the result. Here we call the / with GET and expect an HTTP 200 (thus OK) response with the plain text message of `Hello World, from Spring Boot 2!`.

3-2 Exposing REST Resources with Spring MVC

Problem

You want to use Spring MVC to expose REST based resources `@WebMvcTest`.

Solution

You will need a JSON library to do the JSON marshalling (although you could use XML and other formats as well, as content negotiation[3] is part of REST). In this recipe we will use the Jackson[4] library to take care of the JSON conversion.

How It Works

Imagine you are working for a library and you need to develop a REST API to make it possible to list and search books.

The `spring-boot-starter-web` dependency (see also Recipe 3.1) already includes the needed Jackson libraries by default.

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

Note You can also use the Google GSON library; just use the appropriate GSON dependency instead.

[3]https://www.ics.uci.edu/~fielding/pubs/dissertation/evaluation.htm#sec_6_3_2_7
[4]https://github.com/FasterXML/jackson

As you are making an application for a library it will probably include books, so let's create a Book class.

```
package com.apress.springbootrecipes.library;

import java.util.*;

public class Book {

  private String isbn;
  private String title;
  private List<String> authors = new ArrayList<>();

  public Book() {}

  public Book(String isbn, String title, String... authors) {
    this.isbn = isbn;
    this.title = title;
    this.authors.addAll(Arrays.asList(authors));
  }

  public String getIsbn() {
    return isbn;
  }

  public void setIsbn(String isbn) {
    this.isbn = isbn;
  }

  public String getTitle() {
    return title;
  }

  public void setTitle(String title) {
    this.title = title;
  }

  public void setAuthors(List<String> authors) {
    this.authors = authors;
  }
  public List<String> getAuthors() {
    return Collections.unmodifiableList(authors);
  }

  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Book book = (Book) o;
    return Objects.equals(isbn, book.isbn);
  }

  @Override
  public int hashCode() {
    return Objects.hash(isbn);
  }

  @Override
  public String toString() {
  return String.format("Book [isbn=%s, title=%s, authors=%s]",
                       this.isbn, this.title, this.authors);
  }
}
```

A book is defined by its ISBN number; it has a `title` and 1 or more `authors`.

You would also need a service to work with the books in the library. Let's define an interface and implementation for the `BookService`.

```
package com.apress.springbootrecipes.library;

import java.util.Optional;

public interface BookService {

  Iterable<Book> findAll();
  Book create(Book book);
  Optional<Book> find(String isbn);
}
```

The implementation, for now, is a simple in-memory implementation.

```
package com.apress.springbootrecipes.library;

import org.springframework.stereotype.Service;

import java.util.Map;
import java.util.Optional;
import java.util.concurrent.ConcurrentHashMap;

@Service
class InMemoryBookService implements BookService {

  private final Map<String, Book> books = new ConcurrentHashMap<>();

  @Override
  public Iterable<Book> findAll() {
    return books.values();
  }

  @Override
  public Book create(Book book) {
    books.put(book.getIsbn(), book);
    return book;
  }

  @Override
  public Optional<Book> find(String isbn) {
    return Optional.ofNullable(books.get(isbn));
  }
}
```

The service has been annotated with @Service so that Spring Boot will detect it and create an instance of it.

```
package com.apress.springbootrecipes.library;

import org.springframework.boot.ApplicationRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
```

```java
@SpringBootApplication
public class LibraryApplication {

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

  @Bean
  public ApplicationRunner booksInitializer(BookService bookService) {
    return args -> {
      bookService.create(
        new Book("9780061120084", "To Kill a Mockingbird", "Harper Lee"));
      bookService.create(
        new Book("9780451524935", "1984", "George Orwell"));
      bookService.create(
        new Book("9780618260300", "The Hobbit", "J.R.R. Tolkien"));
    };
  }
}
```

The `LibraryApplication` will detect all the classes and start the server. Upon starting, it will preregister three books so that we have something in our library.

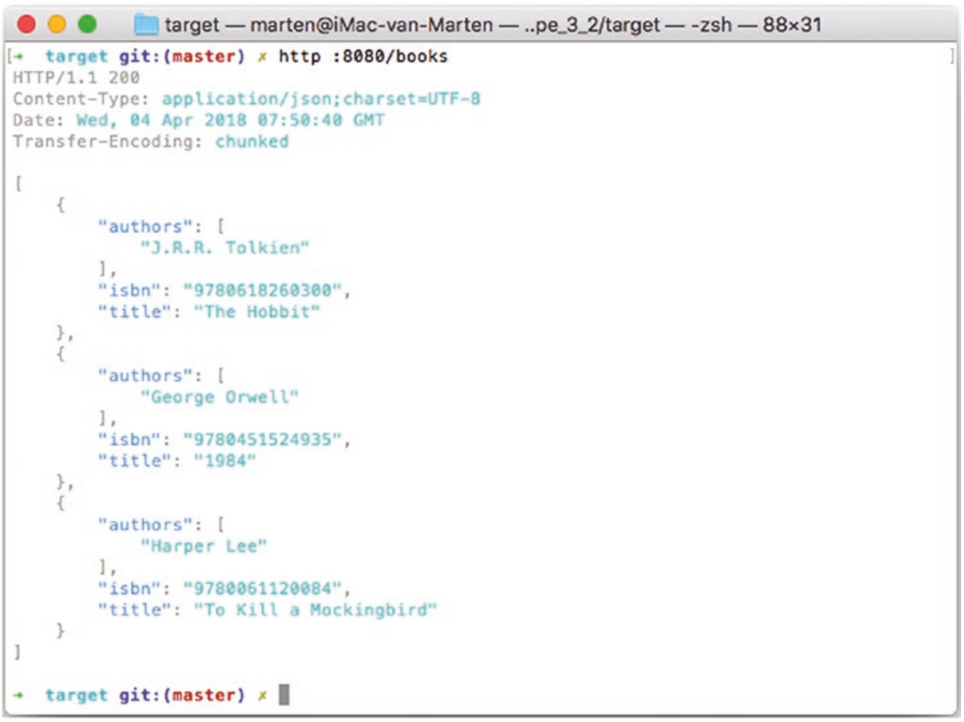
To expose the `Book` as a REST resource, create a class, `BookController`, and annotate it with `@RestController`. Spring Boot will detect this class and create an instance of it. Using `@RequestMapping` (and `@GetMapping` and `@PostMapping`) you can write methods to handle the incoming requests.

Note Instead of `@RestController` you could also use `@Controller` and put `@ResponseBody` on each request handling method. Using `@RestController` will implicitly add `@ResponseBody` to request handling methods.

```java
package com.apress.springbootrecipes.library.rest;

import com.apress.springbootrecipes.library.Book;
import com.apress.springbootrecipes.library.BookService;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.util.UriComponentsBuilder;

import java.net.URI;

@RestController
@RequestMapping("/books")
public class BookController {

  private final BookService bookService;

  public BookController(BookService bookService) {
    this.bookService = bookService;
  }

  @GetMapping
  public Iterable<Book> list() {
    return bookService.findAll();
  }

  @GetMapping("/{isbn}")
  public ResponseEntity<Book> get(@PathVariable("isbn") String isbn) {
    return bookService.find(isbn)
      .map(ResponseEntity::ok)
      .orElse(ResponseEntity.notFound().build());
  }

  @PostMapping
  public Book create(@RequestBody Book book,
                     UriComponentsBuilder uriBuilder) {
    Book created = bookService.create(book);
    URI newBookUri = uriBuilder.path("/books/{isbn}").build(created.getIsbn());
    return ResponseEntity.created(newBookUri).body(created);
  }
}
```

The controller will be mapped to the `/books` path due to the `@RequestMapping ("/books")` annotation on the class. The `list` method will be invoked for GET requests on `/books`. When `/books/<isbn>` is called with a GET request, the `get` method will be invoked and return the result for a single book or, when no book can be found, a 404 response status. Finally, you can add books to the library using a POST request on `/books`; then the `create` method will be invoked and the body of the incoming request will be converted into a book.

After the application has started, you can use HTTPie[5] or cURL[6] to retrieve the books. When using `HTTPie` to access `http://localhost:8080/books`, you should see output similar to that of Figure 3-3.


target — marten@iMac-van-Marten — ..pe_3_2/target — -zsh — 88×31

```
→ target git:(master) ✗ http :8080/books
HTTP/1.1 200
Content-Type: application/json;charset=UTF-8
Date: Wed, 04 Apr 2018 07:50:40 GMT
Transfer-Encoding: chunked

[
    {
        "authors": [
            "J.R.R. Tolkien"
        ],
        "isbn": "9780618260300",
        "title": "The Hobbit"
    },
    {
        "authors": [
            "George Orwell"
        ],
        "isbn": "9780451524935",
        "title": "1984"
    },
    {
        "authors": [
            "Harper Lee"
        ],
        "isbn": "9780061120084",
        "title": "To Kill a Mockingbird"
    }
]

→ target git:(master) ✗
```

Figure 3-3. *JSON output for list of books*

[5] https://httpie.org
[6] https://curl.haxx.se

A request to `http://localhost:8080/books/9780451524935` will give you the result of a single book, in this case for **1984** by George Orwell. Using an unknown ISBN will result in a 404.

When issuing a POST request, we could add a new book to the list.

```
http POST :8080/books \
  title="The Lord of the Rings" \
  isbn="9780618640157" \
  authors:='["J.R.R. Tolkien"]'
```

The result of this call, when done correctly, is the freshly added book and a location header. Now when you get the list of books it should contain four books instead of the three books you started with.

What happens is that HTTPie translates the parameters into a JSON request body, which in turn is read by the Jackson library and turned into a `Book`.

```
  {
    "title": "The Lord of the Rings",
    "isbn": "9780618640157",
    "authors": ["J.R.R. Tolkien"]
  }
```

By default Jackson will use the getter and setter to map the JSON to an object. What happens is that a new `Book` instance is created using the default no-args constructor and all properties are set through the setters. For the `title` property, the `setTitle` is called, etc.

Testing a @RestController

As you want to make sure that the controller does what it is supposed to do, write a test to verify the correct behavior of the controller.

```
Package com.apress.springbootrecipes.library.rest;

import com.apress.springbootrecipes.library.Book;
import com.apress.springbootrecipes.library.BookService;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.result.MockMvcResultMatchers;

import java.util.Arrays;
import java.util.Optional;

import static org.hamcrest.Matchers.*;
import static org.mockito.ArgumentMatchers.anyString;
import static org.mockito.Mockito.when;
import static org.springframework.test.web.servlet.request.
MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.
MockMvcResultMatchers.status;

@RunWith(SpringRunner.class)
@WebMvcTest(BookController.class)
public class BookControllerTest {

  @Autowired
  private MockMvc mockMvc;

  @MockBean
  private BookService bookService;

  @Test
  public void shouldReturnListOfBooks() throws Exception {

    when(bookService.findAll()).thenReturn(Arrays.asList(
      new Book("123", "Spring 5 Recipes", "Marten Deinum", "Josh Long"),
      new Book("321", "Pro Spring MVC", "Marten Deinum", "Colin Yates")));

    mockMvc.perform(get("/books"))
      .andExpect(status().isOk())
      .andExpect(jsonPath("$", hasSize(2)))
      .andExpect(jsonPath("$[*].isbn", containsInAnyOrder("123", "321")))
      .andExpect(jsonPath("$[*].title",
        containsInAnyOrder("Spring 5 Recipes", "Pro Spring MVC")));
  }
  @Test
  public void shouldReturn404WhenBookNotFound() throws Exception {

    when(bookService.find(anyString())).thenReturn(Optional.empty());

    mockMvc.perform(get("/books/123")).andExpect(status().isNotFound());
  }

  @Test
  public void shouldReturnBookWhenFound() throws Exception {

    when(bookService.find(anyString())).thenReturn(
     Optional.of(
      new Book("123", "Spring 5 Recipes", "Marten Deinum", "Josh Long")));

    mockMvc.perform(get("/books/123"))
      .andExpect(status().isOk())
      .andExpect(jsonPath("$.isbn", equalTo("123")))
      .andExpect(jsonPath("$.title", equalTo("Spring 5 Recipes")));
  }
}
```

The test uses `@WebMvcTest` to create a `MockMvc`-based test and will create a minimal Spring Boot application to be able to run the controller. The controller needs an instance of a `BookService`, so we let the framework create a mock for this using the `@MockBean` annotation. In the different test methods, we mock the expected behavior (like returning a list of books, returning an empty `Optional`, etc.).

Note Spring Boot uses Mockito [7] to create mocks using @MockBean.

Furthermore, the test uses the JsonPath[8] library so that you can use expressions to test the JSON result. JsonPath is for JSON what Xpath is for XML.

[7] https://site.mockito.org
[8] https://github.com/json-path/JsonPath

3-3 Using Thymeleaf with Spring Boot

Problem

You want to use Thymeleaf to render the pages of your application.

Solution

Add the dependency for Thymeleaf and create a regular `@Controller` to determine the view and fill the model.

How It Works

To get started you will first need to add the `spring-boot-starter-thymeleaf` as a dependency to your project to get the desired Thymeleaf[9] dependencies.

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

With the addition of this dependency, you will get the Thymeleaf library as well as the Thymeleaf Spring Dialect so that the two integrate nicely. Due to the existence of these two libraries, Spring Boot will automatically configure the `ThymeleafViewResolver`.

The `ThymeleafViewResolver` requires a Thymeleaf `ItemplateEngine` to be able to resolve and render the views. A special `SpringTemplateEngine` will be preconfigured with the `SpringDialect` so that you can use SpEL inside Thymeleaf pages.

To configure Thymeleaf, Spring Boot exposes several properties in the `spring.thymeleaf` namespace (Table 3-2).

[9] https://www.thymeleaf.org

Table 3-2. Thymeleaf Properties

Property	Description
`spring.thymeleaf.prefix`	The `prefix` to use for the `ViewResolver`, default `classpath:/templates/`
`Spring.thymeleaf.suffix`	The `suffix` to use for the `ViewResolver`, default `.html`
`spring.thymeleaf.encoding`	The encoding of the templates, default `UTF-8`
`spring.thymeleaf.check-template`	Check if the template exists before rendering, default `true`.
`Spring.thymeleaf.check-template-location`	Check if the template location exists, the default is `true`.
`Spring.thymeleaf.mode`	Thymeleaf `TemplateMode` to use, default `HTML`
`Spring.thymeleaf.cache`	Should resolved templates be cached or not, default `true`
`Spring.thymeleaf.template-resolver-order`	Order of the `ViewResolver` default is 1.
`Spring.thymeleaf.view-names`	The view names (comma separated) that can be resolved with this `ViewResolver`
`spring.thymeleaf.excluded-view-names`	The view names (comma separated) that are excluded from being resolved
`Spring.thymeleaf.enabled`	Should Thymeleaf be enabled, default `true`
`Spring.thymeleaf.enable-spring-el-compiler`	Enable the compilation of SpEL expressions, default `false`
`Spring.thymeleaf.servlet.content-type`	Content-Type value used to write the HTTP Response, default `text/html`

Adding an Index Page

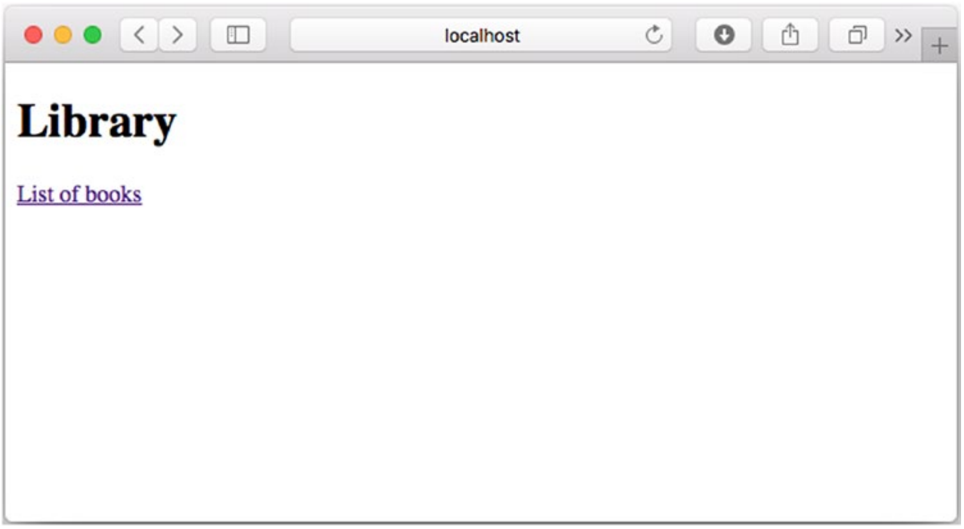
First add an index page to the application. Create an `index.html` inside the `src/main/resources/templates` directory (the default location).

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

```
<head>
    <meta charset="UTF-8">
    <title>Spring Boot Recipes - Library</title>
</head>
<body>

<h1>Library</h1>

<a th:href="@{/books.html}" href="#">List of books</a>

</body>
</html>
```

This is just a basic HTML5 page with some minor additions for Thymeleaf. First there is `xmlns:th="http://www.thymeleaf.org"` to enable the namespace for Thymeleaf. The namespace is used in the link through `th:href`. The `@{/books.html}` will be expanded, by Thymeleaf, to a proper link and placed in the actual `href` attribute of the link.

Now when running the application and visiting the homepage (`http://localhost:8080/`), you should be greeted by a page with a link to the books overview (Figure 3-4).



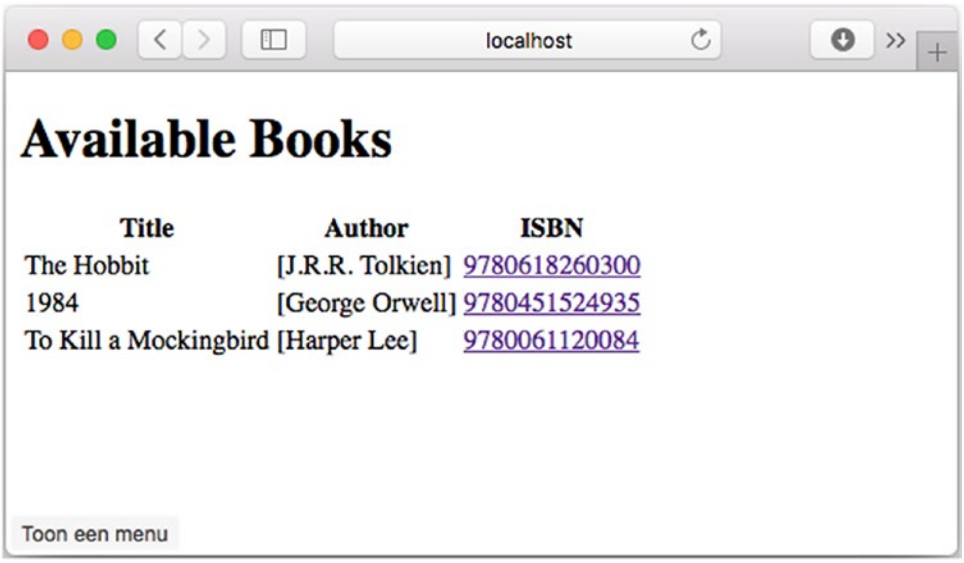
Library

List of books

Figure 3-4. Rendered index page

Adding a Controller and View

When clicking the link provided in the index page, we want to be shown a page that shows a list of available books in the library (Figure 3-5). For this, two things need to be added: first a controller that can handle the request and prepare the model, and second a view to render the list of books.



Available Books

Title	Author	ISBN
The Hobbit	[J.R.R. Tolkien]	9780618260300
1984	[George Orwell]	9780451524935
To Kill a Mockingbird	[Harper Lee]	9780061120084

Figure 3-5. *Books list page*

Let's add a controller that will fill the model with a list of books and select the name of the view to render. A controller is a class annotated with `@Controller` and which contains request handling methods (methods annotated with `@RequestMapping` or as in this recipe `@GetMapping`, which is a specialized `@RequestMapping` annotation).

```
package com.apress.springbootrecipes.library.web;

@Controller
public class BookController {

  private final BookService bookService;

  public BookController(BookService bookService) {
    this.bookService = bookService;
  }
  @GetMapping("/books.html")
  public String all(Model model) {
    model.addAttribute("books", bookService.findAll());
    return "books/list";
  }
}
```

The BookController needs the BookService so that it can obtain a list of books to show. The all method has an org.springframework.ui.Model as method argument so that we can put the list of books in the model. A request handling method can have different arguments[10]; one of them is the Model class. In the all method, we use the BookService to retrieve all the books from the datastore and add it to the model using model.addAttribute. The list of books is now available in the model under the key books.

Finally, we return the name of the view to render books/list. This name is passed on to the ThymeleafViewResolver and will result in a path to classpath:/templates/books/list.html.

Now that the controller together with the request handling method has been added, we need to create the view. Create a list.html in the src/main/templates/books directory.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head lang="en">
  <meta charset="UTF-8">
  <title>Library - Available Books</title>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
</head>
<body>
  <h1>Available Books</h1>
  <table>
    <thead>
      <tr>
        <th>Title</th>
```

[10] https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#mvc-ann-arguments

```
        <th>Author</th>
        <th>ISBN</th>
      </tr>
    </thead>
    <tbody>
      <tr th:each="book : ${books}">
        <td th:text="${book.title}">Title</td>
        <td th:text="${book.authors}">Authors</td>
        <td>
          <a th:href="@{/books.html(isbn=${book.isbn})}" href="#"
             th:text="${book.isbn}">1234567890123</a>
        </td>
      </tr>
    </tbody>
  </table>
</body>
</html>
```

This is again an HTML5 page using the Thymeleaf syntax. The page will render a list of books using the `th:each` expression. It will take all the books from the `books` property in the model and for each book create a row. Each column in the row will contain some text using the `th:text` expression; it will print the title, authors, and ISBN of the book. The final column in the table contains a link to the book details. It constructs a URL using the `th:href` expression. Notice the expression between `()`; this will add the `isbn` request parameter.

When launching the application and clicking the link on the index page, you should be greeted with a page showing the contents of the library as shown in Figure 3-5.

Adding a Details Page

Finally, when clicking the ISBN number in the table, you want a page with details to be shown. The link contains a request parameter named `isbn`, which we can retrieve and use in the controller to find a book. The request parameter can be retrieved through a method argument annotated with `@RequestParam`.

The following method will handle the GET request, map the request parameter to the method argument, and includes the model so that we can add the book to the model.

```
@GetMapping(value = "/books.html", params = "isbn")
public String get(@RequestParam("isbn") String isbn, Model model) {

  bookService.find(isbn)
          .ifPresent(book -> model.addAttribute("book", book));

  return "books/details";
}
```

The controller will render the books/details page. Add the details.html to the src/main/resources/templates/books directory.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head lang="en">
    <meta charset="UTF-8">
    <title>Library - Available Books</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
</head>
<body>
    <div th:if="${book != null}">
        <div>
            <div th:text="${book.title}">Title</div>
            <div th:text="${book.authors}">authors</div>
            <div th:text="${book.isbn}">ISBN</div>
        </div>
    </div>

    <div th:if="${book} == null">
        <h1 th:text="'No book found with ISBN: ' + ${param.isbn}">Not
        Found</h1>
    </div>
</body>
</html>
```

This HTML5 Thymeleaf template will render one of the two available blocks on the page. Either the book has been found and it will display the details, else it will show a not found message. This is achieved by using the `th:if` expression. The `isbn` for the not found message is retrieved from the request parameters using the `param` as a prefix. `${param.isbn}` will get the `isbn` request parameter.

3-4 Handling Exceptions

Problem

You want to customize the default white label error page shown by Spring Boot.

Solution

Add an additional `error.html` as a customized error page, or specific error pages for specific HTTP error codes (i.e., `404.html` and `500.html`).

How It Works

Spring Boot by default comes with the error handling enabled and will show a default error page. This can be disabled in full by setting the `server.error.whitelabel.enabled` property to `false`. When disabled, the exception handling will be handled by the Servlet Container instead of the general exception handling mechanism provided by Spring and Spring Boot.

There are also some other properties that can be used to configure the whitelabel error page, mainly on what is going to be included in the model so that it, optionally, could be used to display. See Table 3-3 for the properties.

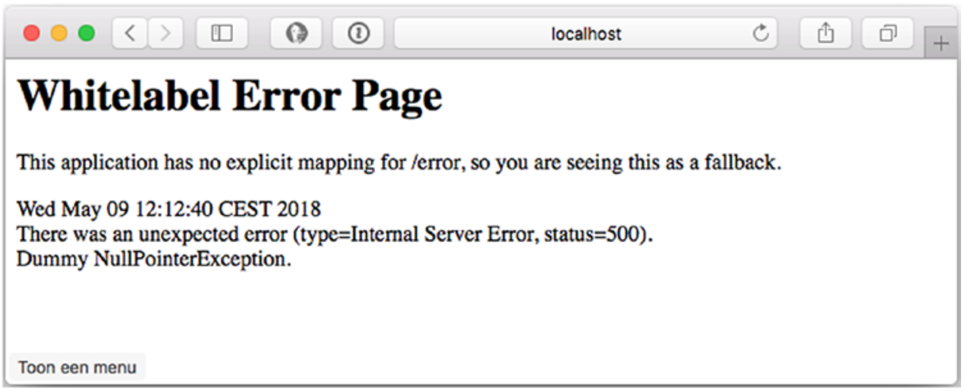
Table 3-3. *Error Handling Properties*

Property	Description
server.error.whitelabel.enabled	Is the whitelabel error page enabled, default `true`
server.error.path	The path of the error page, default `/error`
server.error.include-exception	Should the name of the exception be included in the model, default `false`
server.error.include-stacktrace	Should the stacktrace be included in the model, default `never`

First let's add a method to the `BookController` that forces an exception.

```
@GetMapping("/books/500")
public void error() {
  throw new NullPointerException("Dummy NullPointerException.");
}
```

This will throw an exception and as a result the whitelabel error page will be shown when visiting `http://localhost:8080/books/500` (see Figure 3-6).



Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

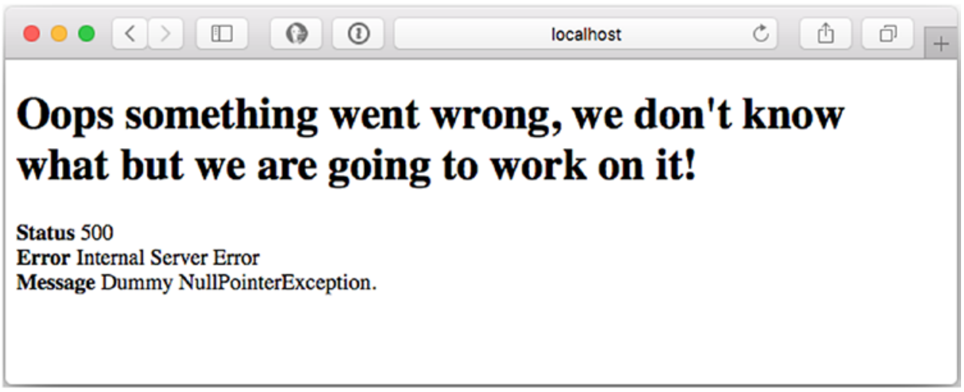
Wed May 09 12:12:40 CEST 2018
There was an unexpected error (type=Internal Server Error, status=500).
Dummy NullPointerException.

Figure 3-6. *Default error page*

This is shown if no error page can be found. To override this, add an `error.html` to the `src/main/resources/templates` directory.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Spring Boot Recipes - Library</title>
</head>
<body>
<h1>Oops something went wrong, we don't know what but we are going to work
on it!</h1>

<div>
    <div>
        <span><strong>Status</strong></span>
        <span th:text="${status}"></span>
    </div>
    <div>
        <span><strong>Error</strong></span>
        <span th:text="${error}"></span>
    </div>
    <div>
        <span><strong>Message</strong></span>
        <span th:text="${message}"></span>
    </div>
    <div th:if="${exception != null}">
        <span><strong>Exception</strong></span>
        <span th:text="${exception}"></span>
    </div>
    <div th:if="${trace != null}">
        <h3>Stacktrace</h3>
        <span th:text="${trace}"></span>
    </div>
</div>
</body>
</html>
```

Now when the application is started and an exception occurs, this custom error page will be shown (Figure 3-7). The page will be rendered by the view technology of your choice (in this case it uses Thymeleaf).

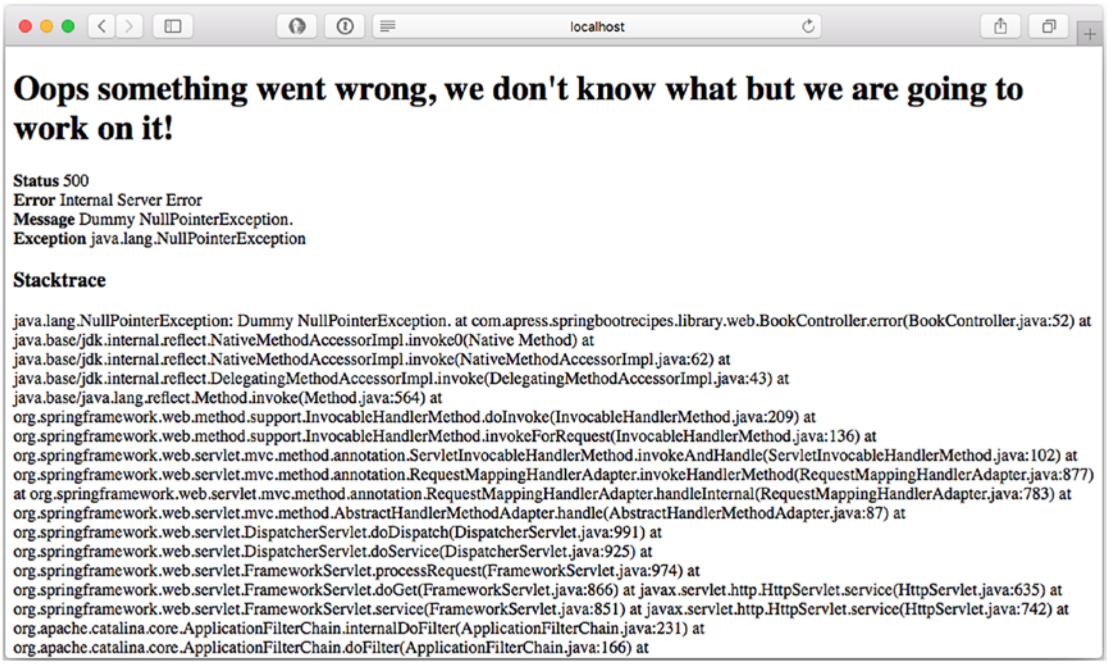


Oops something went wrong, we don't know what but we are going to work on it!

Status 500
Error Internal Server Error
Message Dummy NullPointerException.

Figure 3-7. *Custom error page*

If you now set the `server.error.include-exception` to `true` and `server.error.include-stacktrace` to `always`, the customized error page will also include the classname of the exception and the stacktrace (Figure 3-8).



Oops something went wrong, we don't know what but we are going to work on it!

Status 500
Error Internal Server Error
Message Dummy NullPointerException.
Exception java.lang.NullPointerException

Stacktrace

```
java.lang.NullPointerException: Dummy NullPointerException. at com.apress.springbootrecipes.library.web.BookController.error(BookController.java:52) at
java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method) at
java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62) at
java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43) at
java.base/java.lang.reflect.Method.invoke(Method.java:564) at
org.springframework.web.method.support.InvocableHandlerMethod.doInvoke(InvocableHandlerMethod.java:209) at
org.springframework.web.method.support.InvocableHandlerMethod.invokeForRequest(InvocableHandlerMethod.java:136) at
org.springframework.web.servlet.mvc.method.annotation.ServletInvocableHandlerMethod.invokeAndHandle(ServletInvocableHandlerMethod.java:102) at
org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.invokeHandlerMethod(RequestMappingHandlerAdapter.java:877)
at org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.handleInternal(RequestMappingHandlerAdapter.java:783) at
org.springframework.web.servlet.mvc.method.AbstractHandlerMethodAdapter.handle(AbstractHandlerMethodAdapter.java:87) at
org.springframework.web.servlet.DispatcherServlet.doDispatch(DispatcherServlet.java:991) at
org.springframework.web.servlet.DispatcherServlet.doService(DispatcherServlet.java:925) at
org.springframework.web.servlet.FrameworkServlet.processRequest(FrameworkServlet.java:974) at
org.springframework.web.servlet.FrameworkServlet.doGet(FrameworkServlet.java:866) at javax.servlet.http.HttpServlet.service(HttpServlet.java:635) at
org.springframework.web.servlet.FrameworkServlet.service(FrameworkServlet.java:851) at javax.servlet.http.HttpServlet.service(HttpServlet.java:742) at
org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:231) at
org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166) at
```

Figure 3-8. *Custom error page with stacktrace*

Next to providing a custom generic error page, you could also add an error page for specific HTTP status codes. This can be achieved by adding a `<http-status>.html` to the `src/main/resources/templates/error` directory. Let's add a `404.html` to be shown for unknown URLs.

```html
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Spring Boot Recipes - Library - Resource Not Found</title>
</head>
<body>
<h1>Oops the page couldn't be located.</h1>
</body>
</html>
```

When navigating to a URL that is unknown to the application, it will render this page, and when triggering the exception it will still show the customized error page, as shown in Figures 3-7 and 3-8.

Tip You can also add a `4xx.html` or `5xx.html` for a custom error page for all http status code in the 400 or 500 range.

Adding Attributes to the Model

By default, Spring Boot will include the attributes in the model for the error page, as listed in Table 3-4.

Table 3-4. *Default Error Model Attributes*

Attribute	Description
timestamp	The time that the errors were extracted
status	The status code
error	The error reason
exception	The class name of the root exception (if configured)
message	The exception message
errors	Any `ObjectError` from a `BindingResult` (when using binding and/or validation)
trace	The exception stack trace (if configured)
path	The URL path when the exception was raised

This is all done through the use of an `ErrorAttributes` component. The default used and configured is the `DefaultErrorAttributes`. You can create your own `ErrorAttributes` handler to create a custom model or extend the `DefaultErrorAttributes` to add additional attributes.

```
package com.apress.springbootrecipes.library;

import org.springframework.boot.web.servlet.error.DefaultErrorAttributes;
import org.springframework.web.context.request.WebRequest;
```

```java
import java.util.Map;

public class CustomizedErrorAttributes extends DefaultErrorAttributes {

  @Override
  public Map<String, Object> getErrorAttributes(WebRequest webRequest,
  boolean includeStackTrace) {
    Map<String, Object> errorAttributes =
      super.getErrorAttributes(webRequest, includeStackTrace);
    errorAttributes.put("parameters", webRequest.getParameterMap());
    return errorAttributes;
  }
}
```

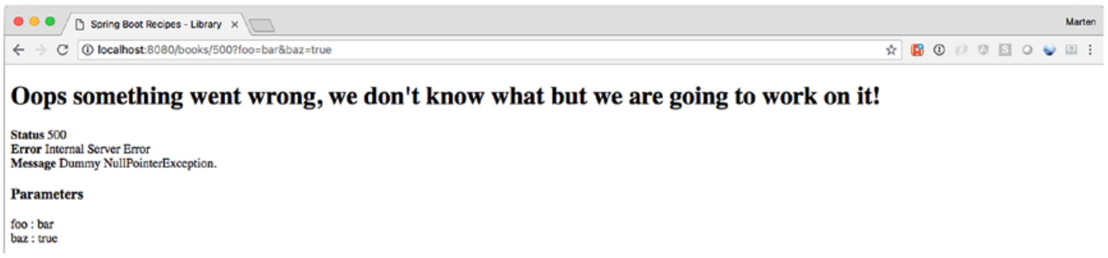
The CustomizedErrorAttributes will add the original request parameters to the model next to the default attributes. Next step is to configure this as a bean in the LibraryApplication. Spring Boot will then detect it and use it instead of configuring the default.

```java
@Bean
public CustomizedErrorAttributes errorAttributes() {
  return new CustomizedErrorAttributes();
}
```

Finally, you might want to use the additional properties in your error.html.

```html
<div th:if="${parameters != null}">
  <h3>Parameters</h3>
  <span th:each="parameter :${parameters}">
    <div th:text="${parameter.key} + ' : ' + ${#strings.
    arrayJoin(parameter.value, ',')}"></div>
    </span>
</div>
```

When the preceding part is included in your error.html it will print the content of the map of parameters available in the model (Figure 3-9).


Spring Boot Recipes - Library
localhost:8080/books/500?foo=bar&baz=true
Marten

Oops something went wrong, we don't know what but we are going to work on it!

Status 500
Error Internal Server Error
Message Dummy NullPointerException.

Parameters

foo : bar
baz : true

Figure 3-9. *Custom error page with parameters*

3-5 Internationalizing Your Application

Problem

When developing an internationalized web application, you have to display your web pages in a user's preferred locale. You don't want to create different versions of the same page for different locales.

Solution

To avoid creating different versions of a page for different locales, you should make your web page independent of the locale by externalizing locale-sensitive text messages. Spring is able to resolve text messages for you by using a message source, which has to implement the `MessageSource` interface. In your page templates you can then use either special tags or do lookups for the messages.

How It Works

Spring Boot automatically configures a `MessageSource` when it finds a `messages.properties` in `src/main/resources` (the default location). This `messages.properties` contains the default messages to be used in your application. Spring Boot will use the `Accept-Language` header from the request to determine which locale to use for the current request (see Recipe 3.6 on how to change that).

There are some properties that change the way the `MessageSource` reacts to missing translations, caching, etc. See Table 3-5 for an overview of the properties.

Table 3-5. I18N Properties

Property	Description
`spring.messages.basename`	Comma-separated list of basenames, default `messages`
`spring.messages.encoding`	Message bundle encoding, default `UTF-8`
`spring.messages.always-use-message-format`	Should `MessageFormat` be applied to all messages, default `false`
`spring.messages.fallback-to-system-locale`	Fallback to the systems locale when no resource bundle for the detected locale can be found. When disabled, will load the defaults from the default file. Default `true`
`spring.messages.use-code-as-default-message`	Use the message code as a default message when no message can be found instead of throwing a `NoSuchMessageException`. Default `false`
`spring.messages.cache-duration`	Cache duration, default forever

Tip It can be useful to set the `spring.messages.fallback-to-system-locale` to `false` when deploying your application to the cloud or other external hosting parties. That way you control what the default language of your application is, instead of the (out of your control) environment you are deploying on.

Add a `messages.properties` to the `src/main/resources` directory.

```
main.title=Spring Boot Recipes - Library

index.title=Library
index.books.link=List of books

books.list.title=Available Books
books.list.table.title=Title
books.list.table.author=Author
books.list.table.isbn=ISBN
```

Now change the templates to use the translations; following is the changed `index.html` file.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title th:text="#{main.title}">Spring Boot Recipes - Library</title>
</head>
<body>

<h1 th:text="#{index.title}">Library</h1>

<a th:href="@{/books.html}" href="#" th:text="#{index.books.link}">List of
books</a>

</body>
</html>
```

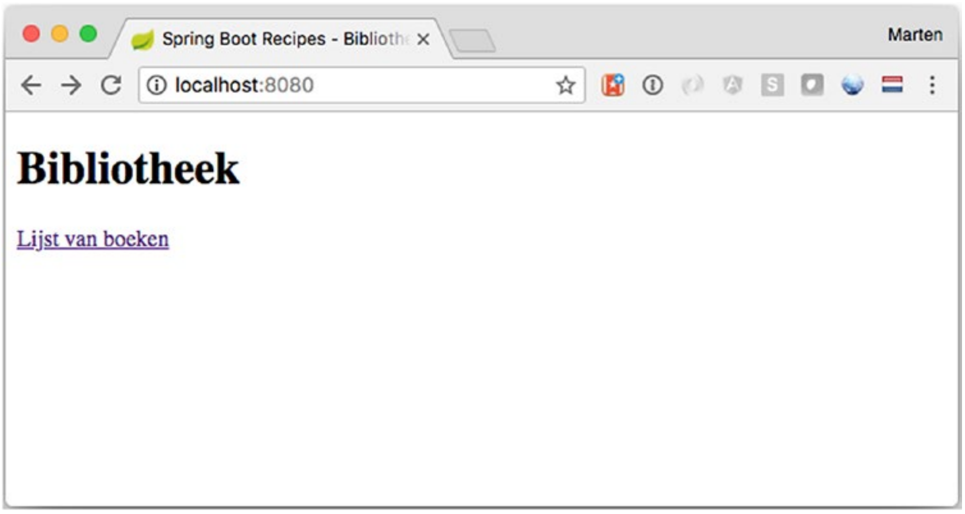
For Thymeleaf you can use a `#{...}` expression in the `th:text` attribute; this will (due to the automatic Spring integration) resolve the messages from the `MessageSource`. When restarting the application, it appears as nothing has changed in the output. However all the texts now come from the `messages.properties`.

Now let's add a `messages_nl.properties` for the Dutch translation of the website.

```
main.title=Spring Boot Recipes - Bibliotheek

index.title=Bibliotheek
index.books.link=Lijst van boeken

books.list.title=Beschikbare Boeken
books.list.table.title=Titel
books.list.table.author=Auteur
books.list.table.isbn=ISBN
```

Now when changing the accept header to Dutch, the website will translate to Dutch (Figure 3-10).

Tip Changing the language for your browser might not be that easy, for Chrome and Firefox there are plug-ins that allow you to switch the `Accept-Language` header easily.



Bibliotheek

Lijst van boeken

Figure 3-10. *Homepage in Dutch*

3-6 Resolving User Locales

Problem

In order for your web application to support internationalization, you have to identify each user's preferred locale and display contents according to this locale.

Solution

In a Spring MVC application, a user's locale is identified by a locale resolver, which has to implement the `LocaleResolver` interface. Spring MVC comes with several `LocaleResolver` implementations for you to resolve locales by different criteria. Alternatively, you may create your own custom locale resolver by implementing this interface.

Spring Boot allows you to set the `spring.mvc.locale-resolver` property. This can be set to `ACCEPT` (the default) or `FIXED`. The first will create an `AcceptHeaderLocaleResolver`; the latter, a `FixedLocaleResolver`.

You can also define a locale resolver by registering a bean of type `LocaleResolver` in the web application context. You must set the bean name of the locale resolver to `localeResolver` so it can be autodetected.

How It Works

Spring MVC ships with several default implementations of the `LocaleResolver` interface. There is a `HandlerInterceptor` provided to allow users to override the locale they want to use, the `LocaleChangeInterceptor`.

Resolving Locales by an HTTP Request Header

The default locale resolver registered by Spring Boot is the `AcceptHeaderLocaleResolver`. It resolves locales by inspecting the `Accept-Language` header of an HTTP request. This header is set by a user's web browser according to the locale setting of the underlying operating system.

Note The `AcceptHeaderLocaleResolver` cannot change a user's locale, because it is unable to modify the locale setting of the user's operating system.

```
@Bean
public LocaleResolver localeResolver () {
  return new AcceptHeaderLocaleResolver();
}
```

Resolving Locales by a Session Attribute

Another option of resolving locales is by `SessionLocaleResolver`. It resolves locales by inspecting a predefined attribute in a user's session. If the session attribute doesn't exist, this locale resolver determines the default locale from the `Accept-Language` HTTP header.

```
@Bean
public LocaleResolver localeResolver () {
    SessionLocaleResolver localeResolver = new SessionLocaleResolver();
    localeResolver.setDefaultLocale(new Locale("en"));
    return localeResolver;
}
```

You can set the `defaultLocale` property for this resolver in case the session attribute doesn't exist. Note that this locale resolver is able to change a user's locale by altering the session attribute that stores the locale.

Resolving Locales by a Cookie

You can also use `CookieLocaleResolver` to resolve locales by inspecting a cookie in a user's browser. If the cookie doesn't exist, this locale resolver determines the default locale from the `Accept-Language` HTTP header.

```
@Bean
public LocaleResolver localeResolver() {
    return new CookieLocaleResolver();
}
```

The cookie used by this locale resolver can be customized by setting the `cookieName` and `cookieMaxAge` properties. The `cookieMaxAge` property indicates how many seconds this cookie should be persisted. The value `-1` indicates that this cookie will be invalid after the browser is closed.

```
@Bean
public LocaleResolver localeResolver() {
    CookieLocaleResolver cookieLocaleResolver = new CookieLocaleResolver();
    cookieLocaleResolver.setCookieName("language");
    cookieLocaleResolver.setCookieMaxAge(3600);
    cookieLocaleResolver.setDefaultLocale(new Locale("en"));
    return cookieLocaleResolver;
}
```

You can also set the `defaultLocale` property for this resolver in case the cookie doesn't exist in a user's browser. This locale resolver is able to change a user's locale by altering the cookie that stores the locale.

Using a Fixed Locale

The FixedLocaleResolver always returns the same fixed locale. By default it returns the JVM default locale but it can be configured to return a different one by setting the `defaultLocale` property.

```
@Bean
public LocaleResolver localeResolver() {
    FixedLocaleResolver cookieLocaleResolver = new FixedLocaleResolver();
    cookieLocaleResolver.setDefaultLocale(new Locale("en"));
    return cookieLocaleResolver;
}
```

Note The `FixedLocaleResolver` cannot change a user's locale because, as the name implies, it is fixed.

Changing a User's Locale

In addition to changing a user's locale by calling `LocaleResolver.setLocale()` explicitly, you can also apply `LocaleChangeInterceptor` to your handler mappings. This interceptor detects if a special parameter is present in the current HTTP request. The parameter name can be customized with the `paramName` property of this interceptor (default is `locale`). If such a parameter is present in the current request, this interceptor changes the user's locale according to the parameter value.

To be able to change the locale, a `LocaleResolver` that allows change has to be used.

```
@Bean
public LocaleResolver localeResolver() {
  return new CookieLocaleResolver();
}
```

To change the locale, add the `LocaleChangeInterceptor` as a bean and register it as an interceptor; for the latter, use the `addInterceptors` method from the `WebMvcConfigurer`.

Note Instead of adding it to the `@SpringBootApplication` you could also create a specialized `@Configuration` annotated class to register the interceptors. Be aware of **not** adding `@EnableWebMvc` to that class, as that will disable the auto configuration from Spring Boot!

```
@SpringBootApplication
public class LibraryApplication implements WebMvcConfigurer {

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

  @Bean
  public LocaleChangeInterceptor localeChangeInterceptor() {
    return new LocaleChangeInterceptor();
  }

  @Bean
  public LocaleResolver localeResolver() {
    return new CookieLocaleResolver();
  }
}
```

Now add the following snippet to the `index.html`:

```
<h3>Language</h3>
<div>
    <a href="?locale=nl" th:text="#{main.language.nl}">NL</a> |
    <a href="?locale=en" th:text="#{main.language.en}">EN</a>
</div>
```

Add the keys to the `messages.properties` file.

```
main.language.nl=Dutch
main.language.en=English
```

Now when selecting one of the languages, the page will re-render and be shown in the selected language; and when you continue browsing, the remainder of the pages will also be shown in the selected language.

3-7 Selecting and Configuring the Embedded Server

Problem

You want to use Jetty as an embedded container instead of the default Tomcat container.

Solution

Exclude the Tomcat runtime and include the Jetty runtime. Spring Boot will automatically detect if Tomcat, Jetty, or Undertow is on the classpath and configure the container accordingly.

How It Works

Spring Boot has out-of-the-box support for Tomcat, Jetty, and Undertow as embedded servlet containers. By default, Spring Boot uses Tomcat as the container (expressed through the `spring-boot-starter-tomcat` dependency in the `spring-boot-starter-web` artifact). The container can be configured using properties for which some apply to all containers and others to a specific container. The global properties are prefixed with `server.` or `server.servlet` whereas the container ones start with `server.<container>` (where `container` is either `tomcat`, `jetty`, or `undertow`).

General Configuration Properties

There are several general server properties available, as seen in Table 3-6.

Table 3-6. *General Server Properties*

Property	Description
`server.port`	The HTTP server port, default 8080
`server.address`	The IP Address to bind to, default `0.0.0.0` (i.e., all adapters)
`server.use-forward-headers`	Should `X-Forwarded-*` headers be applied to the current request, default not set and uses the default from the selected servlet container

(*continued*)

Table 3-6. (*continued*)

Property	Description
`server.server-header`	Name of the header to be sent the Server name, default empty
`server.max-http-header-size`	Maximum size of a HTTP header, default `0` (unlimited)
`server.connection-timeout`	Timeout for HTTP connectors to wait for the next request before closing. Default is empty leaving it to the container; a value of `-1` means infinite and never timeout.
`server.http2.enabled`	Enable Http2 support if the current container supports it. Default `false`
`server.compression.enabled`	Should HTTP compression be enabled, default `false`
`server.compression.mime-types`	Comma seperated list of MIME types that compression applies to
`server.compression.excluded-user-agents`	Comma separated list of user agents for which compression should be disabled
`server.compression.min-response-size`	Minimum size of the request for compression to be applied, default `2048`
`server.servlet.context-path`	The main context path of the application, default launched as the root application
`server.servlet.path`	The path of the main `DispatcherServlet`, default `/`
`server.servlet.application-display-name`	Name used as display name in the container, default `application`
`server.servlet.context-parameters`	Servlet Container Context/init parameters

As the embedded containers all adhere to the Servlet specification, there is also support for JSP pages and that support is enabled by default. Spring Boot makes it easy to change the JSP provider or even disable the support in full. See Table 3-7 for the exposed properties.

Table 3-7. *JSP Related Server Properties*

Property	Description
`server.servlet.jsp.registered`	Should the JSP servlet be registered, default `true`
`server.servlet.jsp.class-name`	The JSP servlet classname, default `org.apache.jasper.servlet.JspServlet` as both Tomcat and Jetty use Jasper as the JSP implementation
`server.servlet.jsp.init-parameters`	Context parameters for the JSP servlet

Note The use of JSP with a Spring Boot application is discouraged and limited.[11]

When using Spring MVC, you might want to use the HTTP Session to store attributes (generally with Spring Security to store CSFR tokens, etc.). The general servlet configuration also allows you to configure the http session and the way it will be stored (cookie, URL, etc.). See Table 3-8 for the properties.

Table 3-8. *HTTP Session Related Server Properties*

`server.servlet.session.timeout`	Session timeout, default 30 minutes
`server.servlet.session.tracking-modes`	Session tracking modes one or more of `cookie`, `url` and `ssl`. Default empty leaving it to the container
`server.servlet.session.persistent`	Should session data be persisted between restarts, default `false`
`server.servlet.session.cookie.name`	Name of the cookie to store the session identifier. Default empty leaving it to the container default
`server.servlet.session.cookie.domain`	Domain value to use for the session cookie. Default empty leaving it to the container default

(*continued*)

[11]https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-developing-web-applications.html#boot-features-jsp-limitations

Table 3-8. (*continued*)

`server.servlet.session.cookie.path`	Path value to use for the session cookie. Default empty leaving it to the container default
`server.servlet.session.cookie.comment`	Comment to use for the session cookie. Default empty leaving it to the container default
`server.servlet.session.cookie.http-only`	Should the session cookie be http-only accesible, default empty leaving it to the container default
`server.servlet.session.cookie.secure`	Should the cookie be send through SSL only, default empty leaving it to the container default
`server.servlet.session.cookie.max-age`	The lifetime of the session cookie. Default empty leaving it to the container default
`server.servlet.session.session-store-directory.directory`	Name of the directory to use for persistent cookies. Has to be an existing directory

Finally, Spring Boot makes it very easy to configure SSL by exposing a few properties, see Table 3-9 and Recipe 3.8 on how to configure SSL.

Table 3-9. *SSL Related Server Properties*

Property	Description
`server.ssl.enabled`	Should SSL be enabled, default `true`
`server.ssl.ciphers`	Supported SSL ciphers, default empty
`server.ssl.client-auth`	Should SSL client authentication be wanted (`WANT`) or needed `NEED`. Default empty
`server.ssl.protocol`	SSL protocol to use, default `TLS`
`server.ssl.enabled-protocols`	Which SSL protocols are enabled, default empty
`server.ssl.key-alias`	The alias to identify the key in the keystore, default empty
`server.ssl.key-password`	The password to access the key in the keystore, default empty

(*continued*)

Table 3-9. (*continued*)

Property	Description
`server.ssl.key-store`	Location of the keystore, typically a JKS file, default empty
`server.ssl.key-store-password`	Password to access the keystore, default empty
`server.ssl.key-store-type`	Type of the keystore, default empty
`server.ssl.key-store-provider`	Provider of the keystore, default empty
`server.ssl.trust-store`	Location of the truststore
`server.ssl.trust-store-password`	Password to access the truststore, default empty
`server.ssl.trust-store-type`	Type of the truststore, default empty
`server.ssl.trust-store-provider`	Provider of the truststore, default empty

Note All the properties mentioned in the aforementioned tables apply **only** when using an embedded container to run your application. When deploying to an external container (i.e., deploying a WAR file), the settings do not apply!

As we are using Thymeleaf, we could disable the registration of the JSP servlet; and let's change the port and compression as well. Place the following in the `application.properties`:

```
server.port=8081
server.compression.enabled=true
server.servlet.jsp.registered=false
```

Now when (re)starting, your application pages (when large enough) will be compressed and the server will run on port 8081 instead of 8080.

Note There is a difference between `server.servlet.context-path` and `server.servlet.path`. When looking at a URL `http://localhost:8080/books` it consists of several parts. The first is the protocol (generally `http` or `https`); then there is the address and port of the server you want to access, followed by the `context-path` (default `/` deployed at the root), which in turn is followed by the `servlet-path`. The `server.context-path` is the main URL to your application; for instance if we set the `server.context-path=/library` the whole application is available on the `/library` URL. (The `DispatcherServlet` still listens to `/` within the `context-path`). Now if we set the `server.path=/dispatch` we need to use `/library/dispatch/books` to access the books.

Next, if we added a second `DispatcherServlet`, which we configure to have a path of `/services`, that would be accesible through `/library/services`. Both `DispatcherServlets` would be active within the main `context-path` of `/library`.

Changing the Runtime Container

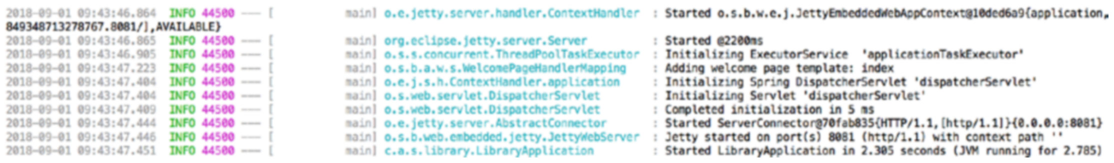
When including the `spring-boot-starter-web` dependency, it will automatically include a dependency to the Tomcat container as it itself has a dependency on the `spring-boot-starter-tomcat` artifact. To enable a different servlet container, the `spring-boot-starter-tomcat` needs to be excluded and one of `spring-boot-starter-jetty` or `spring-boot-starter-undertow` needs to be included.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <exclusions>
    <exclusion>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-tomcat</artifactId>
    </exclusion>
  </exclusions>
```

```
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

In Maven you can use an `<exclusion>` element inside your `<dependency>` to exclude a dependency.

Now when the application is started it will start with Jetty instead of using Tomcat (Figure 3-11).



```
2018-09-01 09:43:46.864  INFO 44500 --- [           main] o.e.jetty.server.handler.ContextHandler  : Started o.s.b.w.e.j.JettyEmbeddedWebAppContext@10ded6a9{application,
849348713278767.8081/],AVAILABLE}
2018-09-01 09:43:46.865  INFO 44500 --- [           main] org.eclipse.jetty.server.Server          : Started @2200ms
2018-09-01 09:43:46.905  INFO 44500 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor  : Initializing ExecutorService  'applicationTaskExecutor'
2018-09-01 09:43:47.223  INFO 44500 --- [           main] o.s.b.a.w.s.WelcomePageHandlerMapping    : Adding welcome page template: index
2018-09-01 09:43:47.404  INFO 44500 --- [           main] o.e.j.s.h.ContextHandler.application     : Initializing Spring DispatcherServlet 'dispatcherServlet'
2018-09-01 09:43:47.404  INFO 44500 --- [           main] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
2018-09-01 09:43:47.409  INFO 44500 --- [           main] o.s.web.servlet.DispatcherServlet        : Completed initialization in 5 ms
2018-09-01 09:43:47.444  INFO 44500 --- [           main] o.e.jetty.server.AbstractConnector       : Started ServerConnector@70fab835{HTTP/1.1,[http/1.1]}{0.0.0.0:8081}
2018-09-01 09:43:47.446  INFO 44500 --- [           main] o.s.b.web.embedded.jetty.JettyWebServer  : Jetty started on port(s) 8081 (http/1.1) with context path ''
2018-09-01 09:43:47.451  INFO 44500 --- [           main] c.a.s.library.LibraryApplication         : Started LibraryApplication in 2.305 seconds (JVM running for 2.785)
```

Figure 3-11. *Bootstrap logging with Jetty container*

3-8 Configuring SSL for the Servlet Container

Problem

You want your application to be accessible through HTTPS next (or instead of) HTTP.

Solution

Get a certificate, place it in a keystore, and use the `server.ssl` namespace to configure the keystore. Spring Boot will then automatically configure the server to be accessible through HTTPS only.

How It Works

Using the `server.ssl.key-store` (and related properties) you can configure the embedded container to only accept HTTPS connection. Before you can configure SSL you will need to have a certificate to secure your application with. Generally you will want to get a certificate from a certification authority like VeriSign or Let's Encrypt. However, for development purposes you can use a self-signed certificate (see the section on **Creating a Self-Signed Certificate**).

Creating a Self-Signed Certificate

Java comes packaged with a tool called the `keytool`, which can be used to generate certificates among other things.

```
keytool -genkey -keyalg RSA -alias sb2-recipes -keystore sb2-recipes.pfx
-storepass password -validity 3600 -keysize 4096 -storetype pkcs12
```

The preceding command will tell `keytool` to generate a key using the RSA algorithm and place it in the keystore named `sb2-recipes.pfx` with the alias `sb2-recipes`, and it will be valid for 3,600 days. When running the command, it will ask a few questions; answer them accordingly (or leave empty). After that, there will be a file called `sb2-recipes.pfx` containing the certificate and protected with a password.

Place this file in the `src/main/resources` folder so that it is packaged as part of your application and Spring Boot can easily access it.

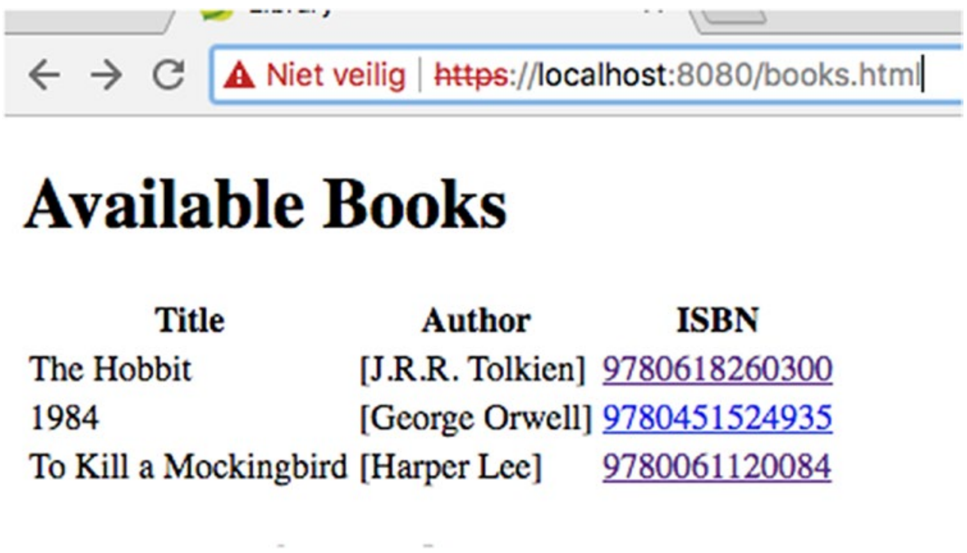
Warning Using a self-signed certificate will produce a warning in the browser that the website isn't safe and protected, because the certificate isn't issued by a trusted authority (see also Figure 3-12).

Configure Spring Boot to Use the Keystore

Spring Boot will need to know about the keystore to be able to configure the embedded container. For this, use the `server.ssl.key-store` property. You will also need to specify the type of keystore (`pkcs12`) and the password.

```
server.ssl.key-store=classpath:sb2-recipes.pfx
server.ssl.key-store-type=pkcs12
server.ssl.key-store-password=password
server.ssl.key-password=password
server.ssl.key-alias=sb2-recipes
```

Now when opening the `http://localhost:8080/books.html` page, it will be served through https (although with a warning). See Figure 3-12.



Available Books

Title	Author	ISBN
The Hobbit	[J.R.R. Tolkien]	9780618260300
1984	[George Orwell]	9780451524935
To Kill a Mockingbird	[Harper Lee]	9780061120084

Figure 3-12. *HTTPS access*

Support Both HTTP and HTTPS

Spring Boot by default only starts one connector: either HTTP or HTTPS but not both. If you want to support both HTTP and HTTPS, you will manually have to add an additional connector. It is easiest to create the HTTP connector yourself and let Spring Boot set up the SSL part.

First let's configure Spring Boot to start the server on port 8443.

```
server.port=8443
```

To add an additional connector to the embedded Tomcat, you will need to add the `TomcatServletWebServerFactory` as a bean to our context. Normally Spring Boot would detect the container and select the `WebServerFactory` to use; however, as a customization needs to be done we need to add it manually. This bean can be added to a `@Configuration` annotated class or to the `LibraryApplication` class.

```
@Bean
public TomcatServletWebServerFactory tomcatServletWebServerFactory() {
  var factory = new TomcatServletWebServerFactory();
  factory.addAdditionalTomcatConnectors(httpConnector());
  return factory;
}
```

```
private Connector httpConnector() {
  var connector = new Connector(TomcatServletWebServerFactory.DEFAULT_PROTOCOL);
  connector.setScheme("http");
  connector.setPort(8080);
  connector.setSecure(false);
  return connector;
}
```

This will add an additional connector on port 8080. The application will now be usable from both port 8080 and 8443. Using Spring Security, you could now force access to parts of your application over HTTPS instead of HTTP.

Tip If you don’t want to explicitly configure the `TomcatServletWebServerFactory` you could also use a `BeanPostProcessor` to register the additional Tomcat `Connector` with the `TomcatServletWebServerFactory`. That way you could implement this for different embedded containers instead of being tied to a single container.

```
@Bean
public BeanPostProcessor addHttpConnectorProcessor() {
  return new BeanPostProcessor() {
    @Override
    public Object postProcessBeforeInitialization(Object bean, String
    beanName)
    throws BeansException {
      if (bean instanceof TomcatServletWebServerFactory) {
        var factory = (TomcatServletWebServerFactory) bean;
        factory.addAdditionalTomcatConnectors(httpConnector());
      }
      return bean;
    }
  };
}
```

Redirect HTTP to HTTPS

Instead of supporting both HTTP and HTTPS, another option is to support only HTTPS and redirect the traffic from HTTP to HTTPS. The configuration is similar to that as when supporting both HTTP and HTTPS. However, you now configure the connector to redirect all traffic from 8080 to 8443.

```
@Bean
public TomcatServletWebServerFactory tomcatServletWebServerFactory() {
  var factory = new TomcatServletWebServerFactory();
  factory.addAdditionalTomcatConnectors(httpConnector());
  factory.addContextCustomizers(securityCustomizer());
  return factory;
}

private Connector httpConnector() {
  var connector = new Connector(TomcatServletWebServerFactory.DEFAULT_PROTOCOL);
  connector.setScheme("http");
  connector.setPort(8080);
  connector.setSecure(false);
  connector.setRedirectPort(8443);
  return connector;
}

private TomcatContextCustomizer securityCustomizer() {
  return context -> {
    var securityConstraint = new SecurityConstraint();
    securityConstraint.setUserConstraint("CONFIDENTIAL");
    var collection = new SecurityCollection();
    collection.addPattern("/*");
    securityConstraint.addCollection(collection);
    context.addConstraint(securityConstraint);
  };
}
```

The `httpConnector` now has a `redirectPort` set so that it knows which port to use. Finally, you need to secure all URLs with a `SecurityConstraint`. With Spring Boot you can use a specialized `TomcatContextCustomizer` to post-process the `Context` from Tomcat before it is started. The constraint makes everything (due to the use of `/*` as pattern) confidential (one of `NONE`, `INTEGRAL`, or `CONFIDENTIAL` is allowed) and the result is that everything will be redirected to https.