



Community Experience Distilled

Mastering Spring MVC 4

Gain expertise in designing real-world web applications using the Spring MVC framework

Geoffroy Warin

[PACKT] open source*

community experience distilled

www.allitebooks.com

Mastering Spring MVC 4

Gain expertise in designing real-world web applications using the Spring MVC framework

Geoffroy Warin



BIRMINGHAM - MUMBAI

Mastering Spring MVC 4

Copyright © 2015 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: September 2015

Production reference: 1080915

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78398-238-7

www.packtpub.com

Credits

Author

Geoffroy Warin

Project Coordinator

Nidhi Joshi

Reviewers

Raymundo Armendariz
Abu S. Kamruzzaman
Jean-Pol Landrain
Wayne Lund

Proofreader

Safis Editing

Indexer

Mariammal Chettiar

Commissioning Editor

Julian Ursell

Production Coordinator

Conidon Miranda

Acquisition Editor

Nadeem Bagban

Cover Work

Conidon Miranda

Content Development Editor

Pooja Nair

Technical Editor

Vivek Pala

Copy Editor

Pranjali Chury

About the Author

Geoffroy Warin has been programming since he was 10. A firm believer in the Software Craftsmanship movement and open source initiatives, he is a developer by choice and conviction. He has been working on the conception of enterprise-level web applications in Java and JavaScript throughout the course of his career.

At ease with both the backend and frontend, Geoffroy has a strong focus on Clean Code and testability. He believes that developers should strive for readable code that delivers constant value to their customers.

Pair programming and mentorship are his primary tools to promote a test-driven development approach and create great software designs.

He also teaches courses on Java web stacks and is a Groovy and Spring enthusiast.

Lately, he has been part of the reviewing team for *Learning Spring Boot* and *Spring Boot Cookbook*, both by Packt Publishing, which cover the latest major additions to the Spring ecosystem.

Check out his blog at <http://geowarin.github.io> and his Twitter account at <https://twitter.com/geowarin> for fresh Spring and JavaScript programming tips.

I'd like to thank Laure, my life partner, who approved my late nights of writing, and my colleagues at Bi-SAM, who have been nothing but supportive of my project.

About the Reviewers

Raymundo Armendariz is a software developer with over 10 years of experience in building software for Java and .NET platforms, but he is currently devoted to JavaScript.

He is the author of a book based on a JavaScirpt micro-famework.

He has been working for the automotive industry for the most part of his professional life. He has worked for companies such as Autozone, Alldata, TRW, and 1A Auto.

He is the author of *Getting Started with Backbone Marionette*, Packt Publishing, which can be found at <https://www.packtpub.com/web-development/getting-started-backbone-marionette>.

I would like to thank my friends for their help and support.

Abu Kamruzzaman is a web programmer and database analyst at The City University of New York. For over 10 years, he has developed and maintained web applications for class grading and registration that can be used by colleges and universities. Since November 2014, he has been working as a PeopleSoft development specialist for CUNY's central office. His current project is about building Data Warehouse for CUNY using OBIEE with the Business Intelligence team. Before joining the central office, he worked at various CUNY campuses since 2001. He also teaches graduate and undergraduate IT courses, instructing the very students who depend on his applications. Since 2001, he has been teaching courses on J2EE, DBMS, data warehouse, object-oriented programming, web design, and web programming. He is a faculty member of the Department of Computer Information Systems at Baruch College's Zicklin School of Business. He has a passion for education and a great interest in open source technologies, such as Hadoop, Hive, Pig, NoSQL databases, Java, cloud computing, and mobile app development. He received his master's degree from Brooklyn College/CUNY and his bachelor's degree in computer science from Binghamton University/SUNY. His web address is <http://faculty.baruch.cuny.edu/akamruzzaman/>.

I want to thank my sweet and beautiful wife, Nicole Woods, Esq., for her constant patience, support, and encouragement in everything I do. Thanks to my sweet parents for their blessings and constant prayers. I would also like to thank the author and the Packt Publishing team for giving me the opportunity to work on this book.

Jean-Pol Landrain holds a BSc degree in software engineering with an orientation in network and real-time and distributed computing since 1998. He gradually became a software architect with more than 17 years of experience in object-oriented programming, in particular with C++, Java/JEE, various application servers, operating systems (Windows and Linux), and related technologies.

He works for Agile Partner, an IT consulting company based in Luxembourg, which is already dedicated to the promotion, education, and application of agile development methodologies since 2006. Over the last 5 years, he has participated in the selection and the validation of tools and technologies targeting the development teams of the European Parliament.

He also collaborated with Packt Publishing to review *HornetQ Messaging Developer's Guide* and with Manning Publishing to review *Docker in Action*, *Git in Practice*, *ActiveMq in Action*, and *Spring in Action, First Edition*.

First, I would like to thank my wife, Marie Smets, and my 9-year-old daughter, Phoebe, for their understanding regarding my passion for technology and the time I dedicate to it. I would also like to thank my friends and colleagues at Agile Partner because a life dedicated to technology would be boring without the fun they bring to it.

Unfortunately, I lost my grandfather, André Landrain, and my grandmother, Hélène Guffens, during the elaboration of this book. My thoughts go out to them and to those of you who have lost loved ones. A big thank you goes to the editorial team at Packt Publishing for their patience with the work that was delayed because of these personal events, particularly Nidhi Joshi, the project coordinator, and Geoffroy Warin, the author of this book. They have done fantastic work together and I have absolutely no doubt that you will appreciate the quality of this book. You may well have the best publication so far on Spring MVC in your hands.

Wayne Lund is a PaaS and field engineer for Pivotal. He has over 25 years of experience in enterprise software development and distributed environments, majorly specializing in Spring, Enterprise Java, Groovy, and Grails and extending to systems built with Smalltalk and C++, always with an emphasis on custom and emerging technologies. His objective is to continue enjoying the next great generation of technology with PaaS along with the new generations of the Spring portfolio. This includes an expertise in the Cloud Native applications built with Spring Boot, Spring Cloud, and Spring XD so as to enable Fast Data, Big Data, social, and mobile.

He is currently working for Pivotal with the intersection of Cloud, Data, and Agile. He previously worked for a Fortune 500 health care company and a large global consulting company for many years.

He has also worked on *Learning Spring Application Development*, Packt Publishing.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	vii
Chapter 1: Setting Up a Spring Web Application in No Time	1
Getting started with Spring Tool Suite	2
Getting started with IntelliJ	8
Getting started with start.Spring.io	9
Getting started with the command line	9
Let's get started	11
The Gradle build	12
Let me see the code!	16
Spring Boot behind the curtains	18
The dispatcher and multipart configuration	19
The view resolver, static resources, and locale configuration	23
Error and encoding configuration	26
Embedded Servlet container (Tomcat) configuration	28
The HTTP port	30
The SSL configuration	30
Other configurations	31
Summary	32
Chapter 2: Mastering the MVC Architecture	33
The MVC architecture	34
MVC critics and best practices	35
Anemic Domain Model	35
Learning from the sources	36
Spring MVC 1-0-1	37
Using Thymeleaf	38
Our first page	40

Spring MVC architecture	42
DispatcherServlet	42
Passing data to the view	43
Spring Expression Language	44
Getting data with a request parameter	45
Enough Hello Worlds, let's fetch tweets!	47
Registering your application	47
Setting up Spring Social Twitter	48
Accessing Twitter	49
Java 8 streams and lambdas	51
Material design with WebJars	52
Using layouts	55
Navigation	57
The check point	61
Summary	62
Chapter 3: Handling Forms and Complex URL Mapping	63
The profile page – a form	63
Validation	71
Customize validation messages	73
Custom annotation for validation	77
Internationalization	78
Changing the locale	79
Translating the application text	82
A list in a form	84
Client validation	88
The check point	90
Summary	91
Chapter 4: File Upload and Error Handling	93
Uploading a file	93
Writing an image to the response	98
Managing upload properties	99
Displaying the uploaded picture	102
Handling file upload errors	104
Translating the error messages	108
Placing the profile in a session	109
Custom error pages	112
URL mapping with matrix variables	114
Putting it together	119
The check point	127
Summary	128

Chapter 5: Crafting a RESTful Application	129
What is REST?	129
Richardson's maturity model	130
Level 0 – HTTP	130
Level 1 – Resources	130
Level 2 – HTTP verbs	131
Level 3 – Hypermedia controls	132
API versioning	133
Useful HTTP codes	134
Client is the king	136
Debugging a RESTful API	138
A JSON formatting extension	138
A RESTful client in your browser	138
httpie	138
Customizing the JSON output	139
A user management API	144
Status codes and exception handling	148
Status code with ResponseEntity	149
Status codes with exceptions	151
Documentation with Swagger	155
Generating XML	157
The check point	158
Summary	160
Chapter 6: Securing Your Application	161
Basic authentication	161
Authorized users	163
Authorized URLs	165
Thymeleaf security tags	167
The login form	169
Twitter authentication	174
Setting up social authentication	175
Explanation	178
Distributed sessions	180
SSL	183
Generating a self-signed certificate	183
The easy way	184
The dual way	184
Behind a secured server	186
The check point	186
Summary	187

Chapter 7: Leaving Nothing to Luck – Unit Tests and Acceptance Tests	189
Why should I test my code?	189
How should I test my code?	190
Test-driven development	192
The unit tests	193
The right tools for the job	194
The acceptance tests	194
Our first unit test	195
Mocks and stubs	199
Mocking with Mockito	199
Stubbing our beans while testing	201
Should I use mocks or stubs?	204
Unit testing REST controllers	204
Testing the authentication	211
Writing acceptance tests	213
The Gradle configuration	213
Our first FluentLenium test	215
Page Objects with FluentLenium	221
Making our tests more Groovy	225
Unit tests with Spock	225
Integration tests with Geb	229
Page Objects with Geb	230
The check point	234
Summary	235
Chapter 8: Optimizing Your Requests	237
A production profile	237
Gzipping	238
Cache control	238
Application cache	240
Cache invalidation	246
Distributed cache	247
Async methods	248
ETags	254
WebSockets	258
The check point	261
Summary	262

Table of Contents

Chapter 9: Deploying Your Web Application to the Cloud	263
Choosing your host	263
Cloud Foundry	264
OpenShift	264
Heroku	265
Deploying your web application to Pivotal Web Services	265
Installing the Cloud Foundry CLI tools	265
Assembling the application	267
Activating Redis	272
Deploying your web application on Heroku	273
Installing the tools	273
Setting up the application	274
Gradle	275
Procfile	276
A Heroku profile	276
Running your application	277
Activating Redis	279
Improving your application	281
Summary	282
Chapter 10: Beyond Spring Web	283
The Spring ecosystem	283
Core	284
Execution	284
Data	284
Other noteworthy projects	285
The deployment	285
Docker	286
Single Page Applications	287
The players	287
The future	288
Going stateless	289
Summary	289
Index	291

Preface

As a web developer, I like to create new things, put them online quickly, and move on to my next idea.

In a world where all our applications are connected to each other, we need to interact with social media to promote our products and complex systems, to provide great value for our users.

Until recently, all this was a distant and complicated world for Java developers. With the birth of Spring Boot and the democratization of cloud platforms, we can now create amazing applications and make them available to everyone in record time, without spending a penny.

In this book, we will build a useful web application from scratch. An application with a lot of neat features, such as internationalization, form validation, distributed sessions and caches, social login, multithreaded programming, and many more.

Also, we will test it completely.

By the end of this book, we will have published our little application and made it available on the Web.

If this sounds like fun to you, let's not waste any more time and get our hands on the code!

What this book covers

Chapter 1, Setting Up a Spring Web Application in No Time, gets us started with Spring Boot really quickly. It covers the tools that will make us more productive, such as Spring Tool Suite and Git. It will also help us to scaffold our application in a snap and see the magic behind Spring boot.

Chapter 2, Mastering the MVC Architecture, guides us through creating a small Twitter search engine. It covers the basics of Spring MVC and the principles of web architecture along the way.

Chapter 3, Handling Forms and Complex URL Mapping, helps you understand how you can create a user profile form. It covers how to validate our data on the server, as well as on the client, and make our application available in different languages.

Chapter 4, File Upload and Error Handling, guides you through adding file upload to your profile form. It demonstrates handling errors properly in a Spring MVC and displaying custom error pages.

Chapter 5, Crafting a RESTful Application, explains the principles of a RESTful architecture. It also helps us to create a user management API accessible through HTTP calls, see which tools can help us design this API, and talks about how we can document it easily.

Chapter 6, Securing Your Application, guides us through securing our application. It covers how we can secure our RESTful API with basic HTTP authentication and our web pages behind a login page. It demonstrates how to enable login via Twitter and store our session on a Redis server to allow our application to scale.

Chapter 7, Leaving Nothing to Luck – Unit Tests and Acceptance Tests, helps us test our application. It discusses testing and TDD, and covers how to unit test our controllers and use modern libraries to design end-to-end tests. It finishes with how Groovy can improve our productivity and the readability of our tests.

Chapter 8, Optimizing Your Requests, takes us through optimizing our application. It covers how to use cache-control and Gzipping. This chapter teaches you how to cache our Twitter search results in-memory and on Redis, and shows you how to multithread the search. As a bonus, implementing Etags and using WebSockets is also covered.

Chapter 9, Deploying Your Web Application to the Cloud, guides us through publishing our application. It shows how the different PaaS solutions can be compared to each other. Then, it demonstrates how to deploy the application on both Cloud Foundry and Heroku.

Chapter 10, Beyond Spring Web, discusses the Spring ecosystem in general, what modern web applications are made of, and where to go from there.

What you need for this book

Although we will build a cutting-edge web application, we do not require you to install a lot of things.

The application that we will build requires Java 8.

You are not forced to, but you definitely should use Git to version control your project. It will be needed if you want to deploy your application on Heroku. Moreover, you will be able to back up your work easily and see the code evolve with the diffs and history. A couple of resources to get started with Git are provided in the first chapter.

I also recommend that you use a good IDE. We will see how to get started quickly with Spring Tool Suite (for free) and IntelliJ Idea (you can obtain a one month trial).

If you have a Mac, you should check Homebrew (<http://brew.sh>). With this package manager, you can install any tool used in this book.

Who this book is for

This book is perfect for developers who are familiar with the fundamentals of Spring programming and are eager to expand their web development skills. Prior knowledge of the Spring framework is recommended.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows:
"You will find the JAR in the directory build/libs."

A block of code is set as follows:

```
public class ProfileForm {  
    private String twitterHandle;  
    private String email;  
    private LocalDate birthDate;  
    private List<String> tastes = new ArrayList<>();  
  
    // getters and setters  
}
```

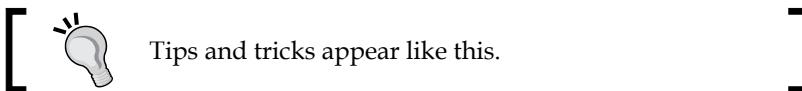
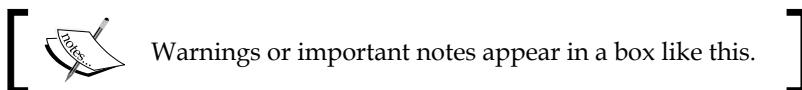
When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
public class ProfileForm {  
    private String twitterHandle;  
    private String email;  
    private LocalDate birthDate;  
    private List<String> tastes = new ArrayList<>();  
  
    // getters and setters  
}
```

Any command-line input or output is written as follows:

```
$ curl https://start.spring.io
```

New terms and important words are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Go to the new project menu and select the **Spring Initializr** project type".



Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

You can also download the example code for this book at <https://github.com/Mastering-Spring-MVC-4/mastering-spring-mvc4>.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Setting Up a Spring Web Application in No Time

In this chapter, we will get straight to the code and set up a web application, which we will be working on for the rest of this book.

We will leverage Spring Boot's autoconfiguration capabilities to build an application with zero boilerplate or configuration files.

I will lay out the big picture regarding how Spring Boot works and how to configure it. There are four ways to get started with Spring:

- Using Spring Tool Suite to generate the starter code
- Using IntelliJ IDEA 14.1, which now has good support for Spring Boot
- Using Spring's website, <http://start.Spring.io>, to download a configurable zip file
- Using the curl command line to <http://start.Spring.io> and achieving the same result

We will use Gradle and Java 8 throughout this book, but don't be scared. Even if you are still working with Maven and a previous version of Java, I bet you will find these technologies easy to work with.

Many official Spring tutorials have both a Gradle build and a Maven build, so you will find examples easily if you decide to stick with Maven. Spring 4 is fully compatible with Java 8, so it would be a shame not to take advantage of lambdas to simplify our code base.

I will also show you some Git commands. I think it's a good idea to keep track of your progress and commit when you are in a stable state. It will also make it easier to compare your work with the source code provided with this book.

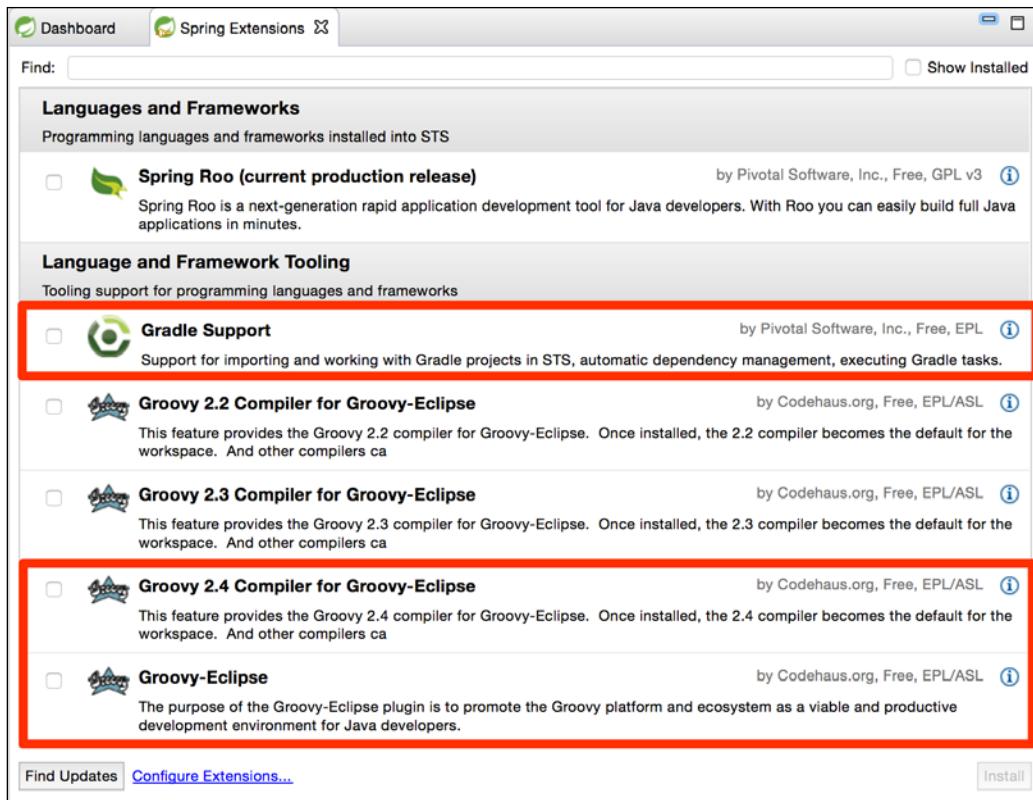
As we will deploy our application with Heroku in *Chapter 9, Deploying Your Web Application to the Cloud*, I recommend that you start versioning your code with Git from the very beginning. I will give you some advice on how to get started with Git later in this chapter.

Getting started with Spring Tool Suite

One of the best ways to get started with Spring and discover the numerous tutorials and starter projects that the Spring community offers is to download **Spring Tool Suite (STS)**. STS is a custom version of eclipse designed to work with various Spring projects, as well as Groovy and Gradle. Even if, like me, you have another IDE that you would rather work with, I strongly recommend that you give STS a shot because it gives you the opportunity to explore Spring's vast ecosystem in a matter of minutes with the "Getting Started" projects.

So, let's visit <https://Spring.io/tools/sts/all> and download the latest release of STS. Before we generate our first Spring Boot project we will need to install the Gradle support for STS. You can find a **Manage IDE Extensions** button on the dashboard. You will then need to download the **Gradle Support** software in the **Language and framework tooling** section.

I also recommend installing the **Groovy Eclipse** plugin along with the **Groovy 2.4 compiler**, as shown in the following screenshot. These will be needed later in this book when we set up acceptance tests with geb:



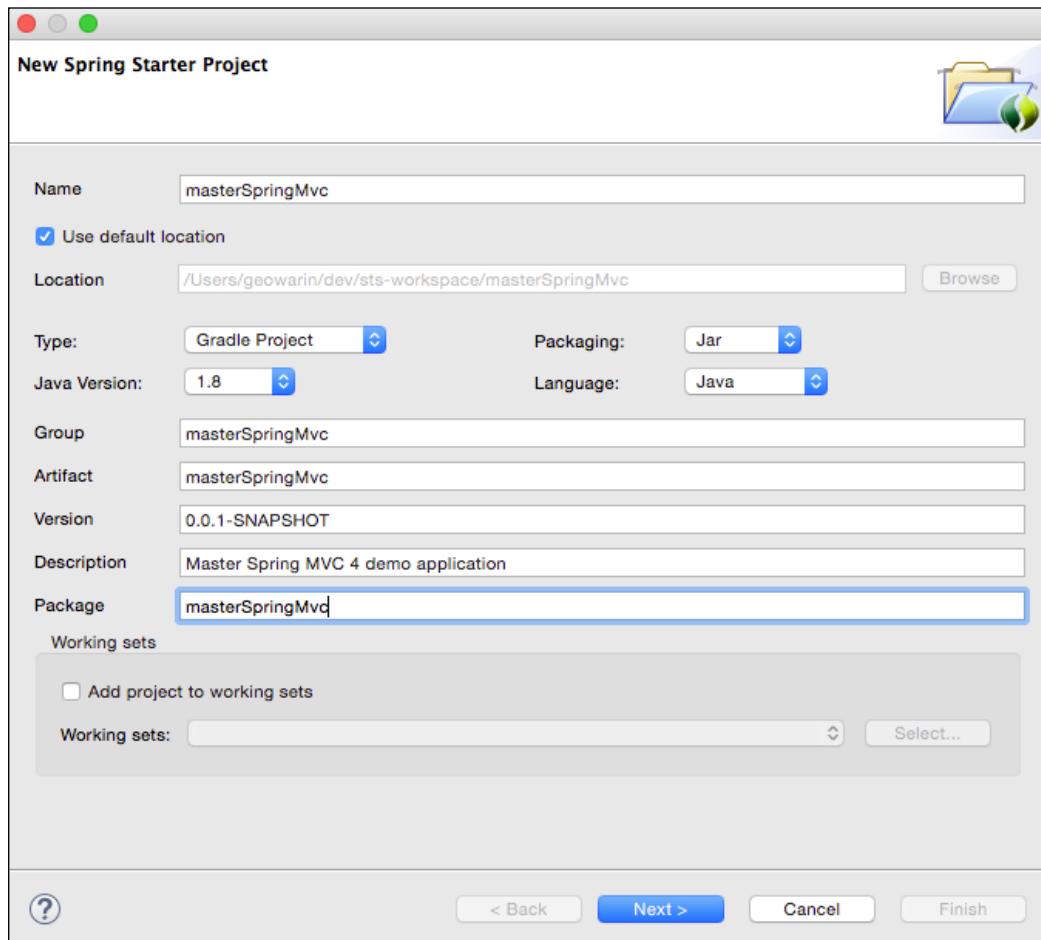
We now have two main options to get started.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

You can also download the example code for this book at <https://github.com/Mastering-Spring-MVC-4/mastering-spring-mvc4>.

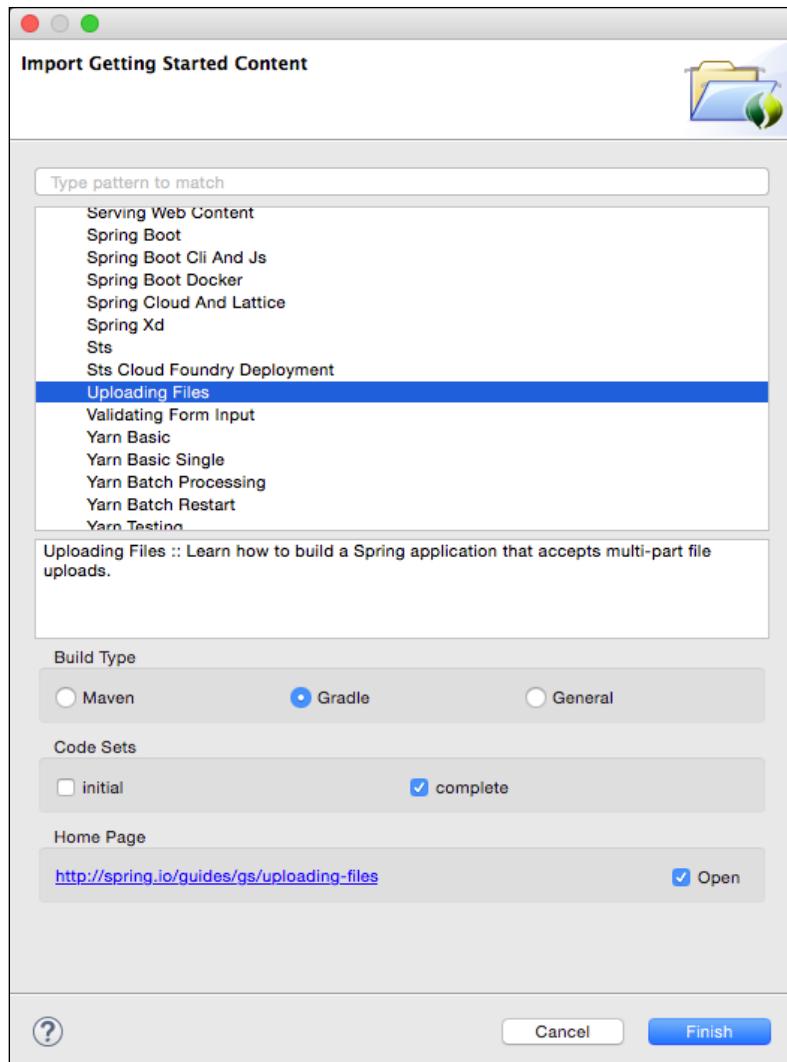
The first option is to navigate to **File | New | Spring Starter Project**, as shown in the following screenshot. This will give you the same options as <http://start.spring.io>, embedded in your IDE:



You also have access to all the tutorials available on <http://spring.io>, selecting in **File | New | Import Getting Started Content** in the top bar. You will have the choice of working with either Gradle or Maven, as shown in the following screenshot:



You can also check out the starter code to follow along with the tutorial, or get the complete code directly.



There is a lot of very interesting content available in the [Getting Started Content](#) and I encourage you to explore it on your own. It will demonstrate the integration of Spring with various technologies that you might be interested in.

For the moment, we will generate a web project as shown in the preceding image. It will be a Gradle application, producing a JAR file and using Java 8.

Here is the configuration we want to use:

Property	Value
Name	masterSpringMvc
Type	Gradle project
Packaging	Jar
Java version	1.8
Language	Java
Group	masterSpringMvc
Artifact	masterSpringMvc
Version	0.0.1-SNAPSHOT
Description	Be creative!
Package	masterSpringMvc

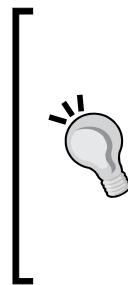
On the second screen you will be asked for the Spring Boot version you want to use and the the dependencies that should be added to the project.

At the time of writing this, the latest version of Spring boot was 1.2.5. Ensure that you always check out the latest release.

The latest snapshot version of Spring boot will also be available by the time you read this. If Spring boot 1.3 isn't released by then, you can probably give it a shot. One of its big features is the awesome devs tools. Refer to <https://spring.io/blog/2015/06/17/devtools-in-spring-boot-1-3> for more details.

At the bottom the configuration window you will see a number of checkboxes representing the various boot starter libraries. These are dependencies that can be appended to your build file. They provide autoconfigurations for various Spring projects.

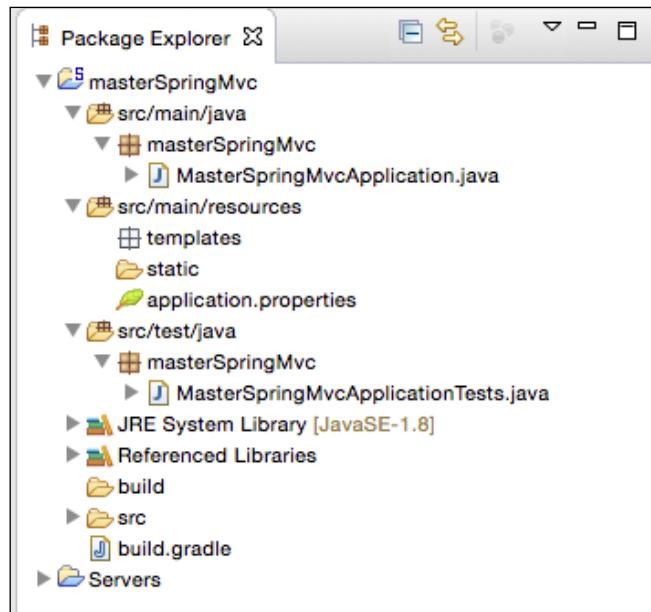
We are only interested in Spring MVC for the moment, so we will check only the Web checkbox.



A JAR for a web application? Some of you might find it odd to package your web application as a JAR file. While it is still possible to use WAR files for packaging, it is not always the recommended practice. By default, Spring boot will create a fat JAR, which will include all the application's dependencies and provide a convenient way to start a web server using Java -jar.

Our application will be packaged as a JAR file. If you want to create a war file, refer to <http://spring.io/guides/gs/convert-jar-to-war/>.

Have you clicked on **Finish** yet? If you have, you should get the following project structure:



We can see our main class **MasterSpringMvcApplication** and its test suite **MasterSpringMvcApplicationTests**. There are also two empty folders, **static** and **templates**, where we will put our static web assets (images, styles, and so on) and obviously our templates (jsp, freemarker, Thymeleaf). The last file is an empty **application.properties** file, which is the default Spring boot configuration file. It's a very handy file and we'll see how Spring boot uses it throughout this chapter.

The **build.gradle** file, the build file that we will detail in a moment.

If you feel ready to go, run the main method of the application. This will launch a web server for us.

To do this, go to the main method of the application and navigate to **Run as | Spring Application** in the toolbar either by right-clicking on the class or clicking on the green play button in the toolbar.

Doing so and navigating to `http://localhost:8080` will produce an error. Don't worry, and read on.

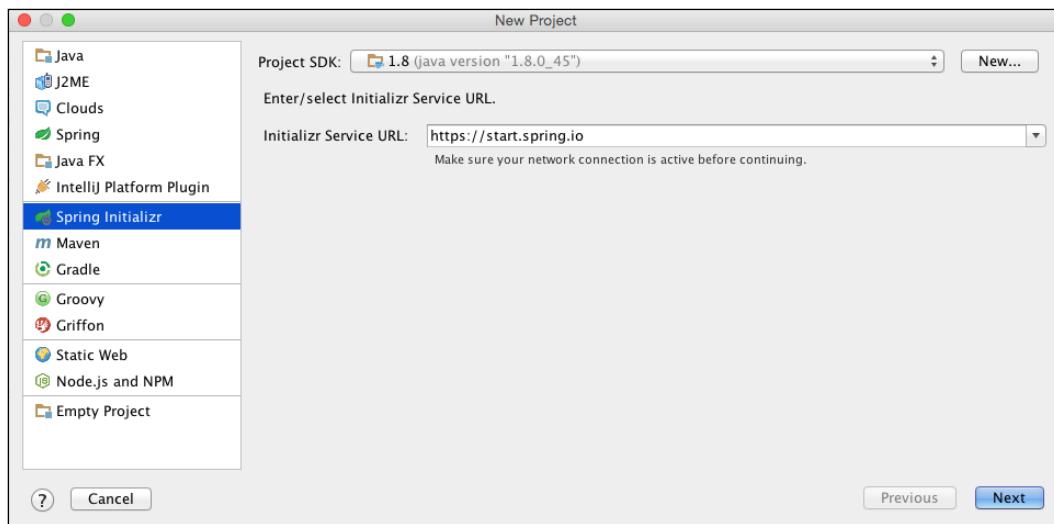
I will show you how to generate the same project without STS, and we will come back to all these files.

Getting started with IntelliJ

IntelliJ IDEA is a very popular tool among Java developers. For the past few years I've been very pleased to pay Jetbrains a yearly fee for this awesome editor.

IntelliJ also has a way of creating Spring boot projects very quickly.

Go to the new project menu and select the **Spring Initializr** project type:

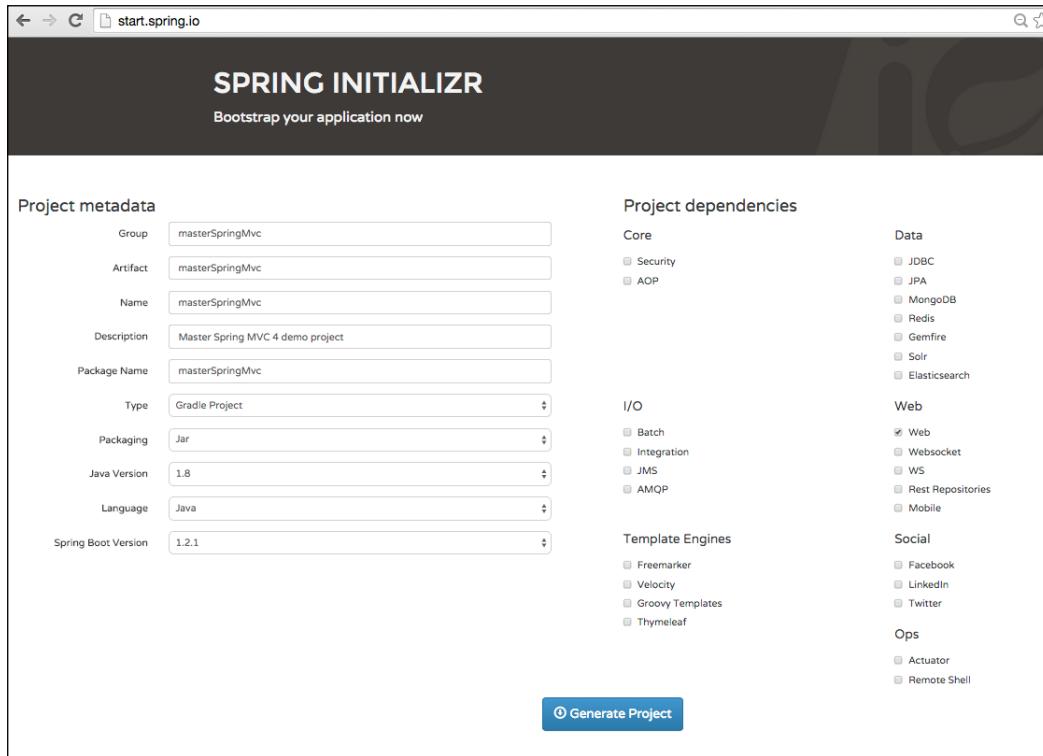


This will give us exactly the same options as STS, so refer to the previous chapter for the detailed configuration.

 You will need to import the Gradle project into IntelliJ. I recommend generating the Gradle wrapper first (refer to the following **Gradle build** section).
If needed, you can reimport the project by opening its `build.gradle` file again.

Getting started with start.Spring.io

Go to <http://start.Spring.io> to get started with start.Spring.io. The system behind this remarkable Bootstrap-like website should be familiar to you! You will see the following screenshot when you go to the previously mentioned link:



Indeed, the same options available with STS can be found here. Clicking on **Generate Project** will download a ZIP file containing our starter project.

Getting started with the command line

For those of you who are addicted to the console, it is possible to curl `http://start.Spring.io`. Doing so will display instructions on how to structure your curl request.

For instance, to generate the same project as earlier, you can issue the following command:

```
$ curl http://start.Spring.io/starter.tgz \
-d name=masterSpringMvc \
-d dependencies=web \
-d language=java \
-d JavaVersion=1.8 \
-d type=gradle-project \
-d packageName=masterSpringMvc \
-d packaging=jar \
-d baseDir=app | tar -xzvf -
% Total    % Received % Xferd  Average Speed   Time     Time     Time
Current                                            Dload  Upload   Total   Spent   Left   Speed
100  1255  100  1119  100    136   1014    123  0:00:01  0:00:01  ---:--:-
- 1015
x app/
x app/src/
x app/src/main/
x app/src/main/Java/
x app/src/main/Java/com/
x app/src/main/Java/com/geowarin/
x app/src/main/resources/
x app/src/main/resources/static/
x app/src/main/resources/templates/
x app/src/test/
x app/src/test/Java/
x app/src/test/Java/com/
x app/src/test/Java/com/geowarin/
x app/build.Gradle
x app/src/main/Java/com/geowarin/AppApplication.Java
x app/src/main/resources/application.properties
x app/src/test/Java/com/geowarin/AppApplicationTests.Java
```

And viola! You are now ready to get started with Spring without leaving the console, a dream come true.



You might consider creating an alias with the previous command, it will help you prototype the Spring application very quickly.



Let's get started

Now that our web application is ready, let's take a look at how it is written. Before going further, we can save our work with Git.

If you don't know anything about Git, I recommend the two following tutorials:

- <https://try.github.io>, which is a good step-by-step interactive tutorial to learn the basic Git commands
- <http://pcottle.github.io/learnGitBranching>, which is an excellent interactive visualization of the Git tree-like structure that will show you basic, as well as very advanced, Git capabilities

Installing Git

 On windows, install Git bash, which can be found at <https://msysgit.github.io>. On Mac, if you use homebrew you should already have Git. Otherwise, use the command `brew install git`. When in doubt, check out the documentation at <https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>.

To version our work with Git, type the following commands in a console:

```
$ cd app  
$ git init
```

With IntelliJ, ignore the generated files: `.idea` and `*.iml`. With eclipse you should commit the `.classpath` and `.settings` folder. In any case you should ignore the `.gradle` folder and the `build` folder.

Create a `.gitignore` file containing the following text:

```
# IntelliJ project files  
.idea  
*.iml  
  
# gradle  
.gradle  
build
```

Now, we can add all the other files to Git:

```
$ git add .
$ git commit -m "Generated with curl start.Spring.io"
[master (root-commit) eded363] Generated with curl start.Spring.io
4 files changed, 75 insertions(+)
create mode 100644 build.Gradle
create mode 100644 src/main/Java/com/geowarin/AppApplication.Java
create mode 100644 src/main/resources/application.properties
create mode 100644 src/test/Java/com/geowarin/AppApplicationTests.Java
```

The Gradle build

If you are unfamiliar with Gradle, think of it as Maven's successor, a modern build tool. Like Maven, it uses conventions such as how to structure a Java application. Our sources will still be found in `src/main/java`, our webapp in `src/main/webapp`, and so on. Not unlike Maven, you can use Gradle plugins to deal with various build tasks. However, Gradle really shines because it allows you to write your own build tasks using the Groovy DSL. The default library makes it easy to manipulate files, declare dependencies between tasks, and execute jobs incrementally.



Installing Gradle

If you're on OS X, you can install Gradle with brew by using `brew install gradle` command. On any *NIX system (Mac included), you can install it with gvm (<http://gvmtool.net/>). Alternatively, you can grab the binary distribution at <https://Gradle.org/downloads>.

The first good practice when creating an application with Gradle is to generate a Gradle wrapper. The Gradle wrapper is a small script that you will share along with your code to ensure that the build will use the same version of Gradle that you used to build the application.

The command to generate the wrapper is `Gradle wrapper`:

```
$ gradle wrapper
```

```
:wrapper
```

```
BUILD SUCCESSFUL
```

```
Total time: 6.699 secs
```

If we look at the new files created, we can see two scripts and two directories:

```
$ git status -s
?? .gradle/
?? gradle/
?? gradlew
?? gradlew.bat
```

The `.gradle` directory contains the Gradle binaries; you wouldn't want to commit those to your version control.

We previously ignored this file along with the build directory so that you could safely `git add` everything else:

```
$ git add .
$ git commit -m "Added Gradle wrapper"
```

The `Gradle` directory contains information on how to get the binaries. The two other files are scripts: a batch script for windows (`Gradlew.bat`) and a shell script for other systems.

We can also run our application with Gradle instead of executing the application from the IDE:

```
$ ./gradlew bootrun
```

Issuing this command will run an embedded tomcat server with our application in it!

The log tells us that the server is running on port 8080. Let's check it out:



I can imagine your disappointment. Our application is not ready for the grand public just yet.

That being said, the work accomplished by the two files our project is made of is rather impressive. Let's review them.

The first one is the Gradle build file, `build.gradle`:

```
buildscript {
    ext {
        springBootVersion = '1.2.5.RELEASE'
    }
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-
plugin:${springBootVersion}")
        classpath("io.spring.gradle:dependency-management-
plugin:0.5.1.RELEASE")
    }
}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'idea'
apply plugin: 'spring-boot'
apply plugin: 'io.spring.dependency-management'

jar {
    baseName = 'masterSpringMvc'
    version = '0.0.1-SNAPSHOT'
}
sourceCompatibility = 1.8
targetCompatibility = 1.8

repositories {
    mavenCentral()
}

dependencies {
    compile("org.springframework.boot:spring-boot-starter-web")
    testCompile("org.springframework.boot:spring-boot-starter-test")
}
```

```

eclipse {
    classpath {
        containers.remove('org.eclipse.jdt.launching.JRE_CONTAINER')
        containers 'org.eclipse.jdt.launching.JRE_CONTAINER/org.
eclipse.jdt.internal.debug.ui.launcher.StandardVMType/JavaSE-1.8'
    }
}

task wrapper(type: Wrapper) {
    gradleVersion = '2.3'
}

```

What do we see here?

- A dependency on the Spring Boot plugin distributed on Maven central.
- Our project is a Java project. IDE project files can be generated by Gradle for IntelliJ or Eclipse.
- The application will generate a JAR file.
- Our project dependencies are hosted on maven central.
- Our classpath includes `spring-boot-starter-web` in production and `spring-boot-starter-test` for testing.
- Some additional configuration for eclipse.
- The version of the Gradle wrapper is 2.3.

The Spring Boot Plugin will generate a fat jar that contains all the dependencies of the project. To build it, type:

```
./gradlew build
```

You will find the JAR in the directory `build/libs`. This directory will contain two files, the fat jar called `masterSpringMvc-0.0.1-SNAPSHOT.jar` and the classic JAR file that does not include any dependencies, `masterSpringMvc-0.0.1-SNAPSHOT.jar.original`.



Runnable jar

One of the main advantages of Spring Boot is embedding everything the application needs in one easily redistributable JAR file, including the web server. If you run `java jar masterSpringMvc-0.0.1-SNAPSHOT.jar`, tomcat will start on port 8080, just like it did when you developed it. This is extremely handy for deploying in production or in the cloud.

Our main dependency here is `spring-boot-starter-web`. Spring Boot provides a good number of starters that will automatically configure some aspects of the application for us by providing typical dependencies and Spring configuration.

For instance, `spring-starter-web` will include dependencies of `tomcat-embedded` and Spring MVC. It will also run the most commonly used Spring MVC configuration and provide a dispatcher listening on the "/" root path, error handling such as the 404 page we saw earlier, and a classical view resolver configuration.

We'll see more on this later. First, let's take a look at the next section.

Let me see the code!

Here is all the code that is needed to run the application. Everything is in a classic main function, which is a huge advantage because you can run your application in your IDE like you would for any other program. You can debug it and also benefit from some class reloading out of the box without a plugin.

This reloading will be available in the debug mode when saving your file in eclipse, or clicking on **Make Project** in IntelliJ. This will be possible only if the JVM is able to switch the new compile version of the class file with the new one; modifying the static variable or touching configuration files will force you to reload the application.

Our main class looks as follows:

```
package masterSpringMvc;

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

@SpringBootApplication
public class AppApplication {

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

Note the `@SpringBootApplication` annotation. If you look at the code of this annotation you will see that it actually combines three other annotations: `@Configuration`, `@EnableAutoConfiguration`, and `@ComponentScan`:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
```

```
@Documented
@Inherited
@Configuration
@EnableAutoConfiguration
@ComponentScan
public @interface SpringApplication {

    /**
     * Exclude specific auto-configuration classes such that they will
     never be applied.
     */
    Class<?>[] exclude() default {};
}
```

The `@Configuration` class should be familiar to you if you've already configured a Spring application with Java code earlier. It indicates that our class will handle classical aspects of a Spring configuration: declaring beans, for instance.

The `@ComponentScan` class is also a classic. It will tell Spring where to look to find our Spring components (services, controllers, and so on). By default, this annotation will scan every current package and everything under it.

The novelty here is `@EnableAutoConfiguration`, which will instruct Spring Boot to do its magic. If you remove it, you will no longer benefit from Spring Boot's autoconfiguration.

The first step when writing an MVC application with Spring Boot is usually to add a controller to our code. Add the controller in the controller subpackage so that it is picked up by the `@ComponentScan` annotation:

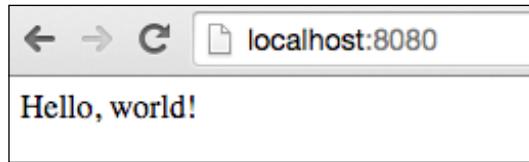
```
package masterSpringMvc.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class HelloController {

    @RequestMapping("/")
    @ResponseBody
    public String hello() {
        return "Hello, world!";
    }
}
```

This time, if you open your browser and visit `http://localhost:8080` you will see this lovely **Hello, world!** output:



Spring Boot behind the curtains

If you already set up a Spring MVC application earlier, you may be used to writing at least a small portion of XML or a handful of Java annotation configuration classes.

Initialization steps are typically as follows:

1. Initializing the DispatcherServlet of Spring MVC.
2. Setting up an encoding filter to ensure that client requests are encoded correctly.
3. Setting up a view resolver to tell Spring where to find our views and in which dialect they are written (jsp, Thymeleaf templates, and so on).
4. Configuring static resources locations (css, js).
5. Configuring supported locales and resource bundles.
6. Configuring a multipart resolver for file uploads to work.
7. Including tomcat or jetty to run our application on a web server.
8. Setting up the error pages (For example 404).

However, Spring Boot handles all that work for us. Because this configuration is typically up to your application, you can come up with an unlimited amount of combinations.

Spring boot, in a way, is an opinionated Spring project configurator. It is based on conventions and will enforce them on your project by default.

The dispatcher and multipart configuration

Let's see what happens behind the curtains.

We will use the default Spring Boot configuration file that was created for us and put it in the debug mode. Add the following line to `src/main/resources/application.properties`:

```
debug=true
```

Now, if we launch our application again we'll see Spring Boot's autoconfiguration report. It is divided into two parts: **positive matches**, which list all autoconfigurations that are used by our application; and **negative matches**, which are Spring Boot autoconfigurations whose requirements weren't met when the application started:

```
=====
AUTO-CONFIGURATION REPORT
=====
```

Positive matches:

```
-----
DispatcherServletAutoConfiguration
- @ConditionalOnClass classes found: org.springframework.web.
servlet.DispatcherServlet (OnClassCondition)
- found web application StandardServletEnvironment
(OnWebApplicationCondition)

EmbeddedServletContainerAutoConfiguration
- found web application StandardServletEnvironment
(OnWebApplicationCondition)

ErrorMvcAutoConfiguration
- @ConditionalOnClass classes found: javax.servlet.Servlet,org.
springframework.web.servlet.DispatcherServlet (OnClassCondition)
- found web application StandardServletEnvironment
(OnWebApplicationCondition)
```

```
HttpEncodingAutoConfiguration
- @ConditionalOnClass classes found: org.springframework.web.
filter.CharacterEncodingFilter (OnClassCondition)
- matched (OnPropertyCondition)
```

```
<Input trimmed>
```

Let's take a closer look at DispatcherServletAutoConfiguration:

```
/**
 * {@link EnableAutoConfiguration} for the Spring
 * {@link DispatcherServlet}. Should work for a standalone application
 * where an embedded
 * servlet container is already present and also for a deployable
 * application using
 * {@link SpringBootServletInitializer}.
 *
 * @author Phillip Webb
 * @author Dave Syer
 */
@Order(Ordered.HIGHEST_PRECEDENCE)
@Configuration
@ConditionalOnWebApplication
@ConditionalOnClass(DispatcherServlet.class)
@AutoConfigureAfter(EmbeddedServletContainerAutoConfiguration.class)
public class DispatcherServletAutoConfiguration {

    /*
     * The bean name for a DispatcherServlet that will be mapped to the
     * root URL "/"
     */
    public static final String DEFAULT_DISPATCHER_SERVLET_BEAN_NAME =
"dispatcherServlet";

    /*
     * The bean name for a ServletRegistrationBean for the
     * DispatcherServlet "/"
     */
    public static final String DEFAULT_DISPATCHER_SERVLET_
REGISTRATION_BEAN_NAME = "dispatcherServletRegistration";
```

```
 @Configuration
 @Conditional(DefaultDispatcherServletCondition.class)
 @ConditionalOnClass(ServletRegistration.class)
 protected static class DispatcherServletConfiguration {

    @Autowired
    private ServerProperties server;

    @Autowired(required = false)
    private MultipartConfigElement multipartConfig;

    @Bean(name = DEFAULT_DISPATCHER_SERVLET_BEAN_NAME)
    public DispatcherServlet dispatcherServlet() {
        return new DispatcherServlet();
    }

    @Bean(name = DEFAULT_DISPATCHER_SERVLET_REGISTRATION_BEAN_
NAME)
    public ServletRegistrationBean dispatcherServletRegistration()
    {
        ServletRegistrationBean registration = new
        ServletRegistrationBean(
            dispatcherServlet(), this.server.
        getServletMapping());
        registration.setName(DEFAULT_DISPATCHER_SERVLET_BEAN_
NAME);
        if (this.multipartConfig != null) {
            registration.setMultipartConfig(this.multipartConfig);
        }
        return registration;
    }

    @Bean
    @ConditionalOnBean(MultipartResolver.class)
    @ConditionalOnMissingBean(name = DispatcherServlet.MULTIPART_
RESOLVER_BEAN_NAME)
    public MultipartResolver multipartResolver(MultipartResolver
resolver) {
        // Detect if the user has created a MultipartResolver but
        // named it incorrectly
        return resolver;
    }
}
```

```
}

@Order(Ordered.LOWEST_PRECEDENCE - 10)
private static class DefaultDispatcherServletCondition extends
SpringBootCondition {

    @Override
    public ConditionOutcome getMatchOutcome(ConditionContext
context,
        AnnotatedTypeMetadata metadata) {
        ConfigurableListableBeanFactory beanFactory = context.
getBeanFactory();
        ConditionOutcome outcome = checkServlets(beanFactory);
        if (!outcome.isMatch()) {
            return outcome;
        }
        return checkServletRegistrations(beanFactory);
    }

}
```

This is a typical Spring Boot configuration class:

- It is annotated with `@Configuration` like any other Spring configuration class.
- It typically declares its priority level with the `@Order` annotation. You can see that `DispatcherServletAutoConfiguration` needs to be configured first.
- It can also contain hints such as `@AutoConfigureAfter` or `@AutoConfigureBefore` to further refine the order in which configurations are processed.
- It is enabled under certain conditions. With `@ConditionalOnClass(DispatcherServlet.class)`, this particular configuration ensures that our classpath contains `DispatcherServlet`, which is a good indication that Spring MVC is in the classpath and the user certainly wants to bootstrap it.

This file also contains classic bean declarations for the Spring MVC dispatcher servlet and a multipart resolver. The whole Spring MVC configuration is broken into multiple files.

It is also worth noting that these beans obey certain rules to check whether they are active. The `ServletRegistrationBean` function will be enabled under the `@Conditional (DefaultDispatcherServletCondition.class)` condition, which is a bit complex but checks whether you already have a dispatcher servlet registered in your own configuration.

The `MultipartResolver` function will become active only if the condition `@ConditionalOnMissingBean(name = DispatcherServlet.MULTIPART_RESOLVER_BEAN_NAME)` is met, for example, if we didn't declare it ourselves.

This means Spring boot only gives you a hand in configuring your application according to common use cases. However, at any point, you can override these defaults and declare your own configuration.

So, the `DispatcherServletAutoConfiguration` class explains why we have a dispatcher servlet and a multipart resolver.

The view resolver, static resources, and locale configuration

Another very relevant piece of configuration is `WebMvcAutoConfiguration`. It declares the view resolver, the locale resolver, and the location of our static resources. The view resolver is as follows:

```
@Configuration
@Import (EnableWebMvcConfiguration.class)
@EnableConfigurationProperties({ WebMvcProperties.class,
    ResourceProperties.class })
public static class WebMvcAutoConfigurationAdapter extends
    WebMvcConfigurerAdapter {

    @Value("${spring.view.prefix:}")
    private String prefix = "";

    @Value("${spring.view.suffix:}")
    private String suffix = "";

    @Bean
    @ConditionalOnMissingBean(InternalResourceViewResolver.class)
    public InternalResourceViewResolver defaultViewResolver() {
        InternalResourceViewResolver resolver = new
        InternalResourceViewResolver();
        resolver.setPrefix(this.prefix);
        resolver.setSuffix(this.suffix);
        return resolver;
    }
}
```

The view resolver configuration is really typical. What's really interesting here is the use of configuration properties to allow users to customize it.

What it says is "I will look for two variables in the user's application.properties called `spring.view.prefix` and `spring.view.suffix`". This is a very handy way to set up the view resolver with only two lines in our configuration.

Keep this in mind for the next chapter. For now, we will just stroll through Spring Boot's code.

Regarding static resources, this configuration includes the following lines:

```
private static final String[] CLASSPATH_RESOURCE_LOCATIONS = {
    "classpath:/META-INF/resources/", "classpath:/resources/",
    "classpath:/static/", "classpath:/public/" };

private static final String[] RESOURCE_LOCATIONS;
static {
    RESOURCE_LOCATIONS = new String[CLASSPATH_RESOURCE_LOCATIONS.length
        + SERVLET_RESOURCE_LOCATIONS.length];
    System.arraycopy(SERVLET_RESOURCE_LOCATIONS, 0, RESOURCE_LOCATIONS,
        0,
        SERVLET_RESOURCE_LOCATIONS.length);
    System.arraycopy(CLASSPATH_RESOURCE_LOCATIONS, 0, RESOURCE_LOCATIONS,
        SERVLET_RESOURCE_LOCATIONS.length,
        CLASSPATH_RESOURCE_LOCATIONS.length);
}

@Override
public void addResourceHandlers(ResourceHandlerRegistry registry) {
    if (!this.resourceProperties.isAddMappings()) {
        logger.debug("Default resource handling disabled");
        return;
    }

    Integer cachePeriod = this.resourceProperties.getCachePeriod();
    if (!registry.hasMappingForPattern("/webjars/**")) {
        registry.addResourceHandler("/webjars/**")
            .addResourceLocations("classpath:/META-INF/resources/
webjars/")
```

```

        .setCachePeriod(cachePeriod) ;
    }
    if (!registry.hasMappingForPattern("/**")) {
        registry.addResourceHandler("/**")
            .addResourceLocations(RESOURCE_LOCATIONS)
            .setCachePeriod(cachePeriod);
    }
}

```

The declaration of resource locations is a bit convoluted but we can still understand two things:

- Any resource accessed with the "webjar" prefix will be resolved inside the classpath inside the classpath. This will allow us to use prepackaged JavaScript dependencies from Maven central.
- Our static resources can reside in any of the locations after our classpath /META-INF/resources/, /resources/, /static/, or /public/.



WebJars are JAR packages of client JavaScript libraries available on Maven central. They include a Maven project file, which allows for transitive dependencies and works in all JVM-based applications. WebJars are an alternative to JavaScript package managers such as bower or npm. They are great for applications that require just a few JavaScript libraries. Find the list of available WebJars on www.webjars.org.

There is also a part of this file that is dedicated to locale management:

```

@Bean
@ConditionalOnMissingBean(LocaleResolver.class)
@ConditionalOnProperty(prefix = "spring.mvc", name = "locale")
public LocaleResolver localeResolver() {
    return new FixedLocaleResolver(
        StringUtils.parseLocaleString(this.mvcProperties.getLocale()));
}

```

This default locale resolver handles only one locale and allows us to define it via the `spring.mvc.locale` configuration property.

Error and encoding configuration

Remember when we first launched our application without adding a controller? We got a funny **Whitelabel Error Page** output.

Error handling is a lot trickier than it looks, especially when you don't have a `web.xml` configuration file and want your application to be portable across web servers. The good news is that Spring Boot takes care of that for us! Let's look at `ErrorMvcAutoConfiguration`:

```
ConditionalOnClass({ Servlet.class, DispatcherServlet.class })
@ConditionalOnWebApplication
// Ensure this loads before the main WebMvcAutoConfiguration so that
the error View is
// available
@AutoConfigureBefore (WebMvcAutoConfiguration.class)
@Configuration
public class ErrorMvcAutoConfiguration implements
EmbeddedServletContainerCustomizer,
Ordered {

    @Value("${error.path:/error}")
    private String errorPath = "/error";

    @Autowired
    private ServerProperties properties;

    @Override
    public int getOrder() {
        return 0;
    }

    @Bean
    @ConditionalOnMissingBean(value = ErrorAttributes.class, search =
SearchStrategy.CURRENT)
    public DefaultErrorAttributes errorAttributes() {
        return new DefaultErrorAttributes();
    }

    @Bean
    @ConditionalOnMissingBean(value = ErrorController.class, search =
SearchStrategy.CURRENT)
    public BasicErrorController basicErrorController(ErrorAttributes
errorAttributes) {
        return new BasicErrorController(errorAttributes);
    }
}
```

```

    @Override
    public void customize(ConfigurableEmbeddedServletContainer
    container) {
        container.addErrorPages(new ErrorPage(this.properties.
    getServletPrefix()
        + this.errorPath));
    }

    @Configuration
    @ConditionalOnProperty(prefix = "error.whitelabel", name =
"enabled", matchIfMissing = true)
    @Conditional(ErrorTemplateMissingCondition.class)
    protected static class WhitelabelErrorViewConfiguration {

        private final SpelView defaultErrorView = new SpelView(
            "<html><body><h1>Whitelabel Error Page</h1>" +
            "<p>This application has no explicit mapping
        for /error, so you are seeing this as a fallback.</p>" +
            "<div id='created'>${timestamp}</div>" +
            "<div>There was an unexpected error
        (type=${error}, status=${status}).</div>" +
            "<div>${message}</div></body></html>");

        @Bean(name = "error")
        @ConditionalOnMissingBean(name = "error")
        public View defaultErrorView() {
            return this.defaultErrorView;
        }

        // If the user adds @EnableWebMvc then the bean name view
        resolver from
        // WebMvcAutoConfiguration disappears, so add it back in to
        avoid disappointment.
        @Bean
        @ConditionalOnMissingBean(BeanNameViewResolver.class)
        public BeanNameViewResolver beanNameViewResolver() {
            BeanNameViewResolver resolver = new
        BeanNameViewResolver();
            resolver.setOrder(Ordered.LOWEST_PRECEDENCE - 10);
            return resolver;
        }

    }
}

```

What does this piece of configuration do?

- It defines a bean, `DefaultErrorAttributes`, which exposes helpful error information via special attributes such as the status, error code, and associated stack trace.
- It defines a `BasicErrorController` bean, which is an MVC controller in charge of displaying the error page we've seen.
- It allows us to deactivate Spring Boot whitelabel error page by setting `error.whitelabel.enabled` to false in our configuration file, `application.properties`.
- We can also leverage our templating engine to provide our own error page. It will be named `error.html`, for example. This is what the condition `ErrorTemplateMissingCondition` checks.

We'll see how to properly handle errors later in this book.

As far as encoding is concerned, the very simple `HttpEncodingAutoConfiguration` function will handle it by providing Spring's `CharacterEncodingFilter` class. It is possible to override the default encoding ("UTF-8") with `spring.http.encoding.charset` and disable this configuration with `spring.http.encoding.enabled`.

Embedded Servlet container (Tomcat) configuration

By default, Spring Boot runs and packages our application using the Tomcat embedded API.

Let's look at `EmbeddedServletContainerAutoConfiguration`:

```
@Order(Ordered.HIGHEST_PRECEDENCE)
@Configuration
@ConditionalOnWebApplication
@Import(EmbeddedServletContainerCustomizerBeanPostProcessorRegistrar.class)
public class EmbeddedServletContainerAutoConfiguration {

    /**
     * Nested configuration for if Tomcat is being used.
     */
    @Configuration
    @ConditionalOnClass({ Servlet.class, Tomcat.class })
    @ConditionalOnMissingBean(value = EmbeddedServletContainerFactory.class, search = SearchStrategy.CURRENT)
```

```
public static class EmbeddedTomcat {  
  
    @Bean  
    public TomcatEmbeddedServletContainerFactory  
tomcatEmbeddedServletContainerFactory() {  
    return new TomcatEmbeddedServletContainerFactory();  
}  
  
}  
  
/**  
 * Nested configuration if Jetty is being used.  
 */  
@Configuration  
@ConditionalOnClass({ Servlet.class, Server.class, Loader.class })  
@ConditionalOnMissingBean(value = EmbeddedServletContainerFactory.  
class, search = SearchStrategy.CURRENT)  
public static class EmbeddedJetty {  
  
    @Bean  
    public JettyEmbeddedServletContainerFactory  
jettyEmbeddedServletContainerFactory() {  
    return new JettyEmbeddedServletContainerFactory();  
}  
  
}  
  
/**  
 * Nested configuration if Undertow is being used.  
 */  
@Configuration  
@ConditionalOnClass({ Servlet.class, Undertow.class,  
SslClientAuthMode.class })  
@ConditionalOnMissingBean(value = EmbeddedServletContainerFactory.  
class, search = SearchStrategy.CURRENT)  
public static class EmbeddedUndertow {  
  
    @Bean  
    public UndertowEmbeddedServletContainerFactory  
undertowEmbeddedServletContainerFactory() {  
    return new UndertowEmbeddedServletContainerFactory();  
}  
  
}  
}
```

The preceding code is pretty straight forward. This code includes three different configurations, which will be activated depending on what's available on your classpath.

You can use Tomcat, tc-server, Jetty, or Undertow with Spring Boot. Your server can be easily replaced by excluding the `spring-boot-starter-tomcat` JAR dependency and replacing it with its Jetty or Undertow equivalent. Please refer to the documentation if you wish to do so.

All the configuration of our Servlet container (Tomcat) will happen in `TomcatEmbeddedServletContainerFactory`. While you should definitely read it because it provides a very advanced configuration of tomcat embedded (for which finding documentation can be hard), we will not look at this class directly.

Instead, I will walk you through the different options available to configure your Servlet Container.

The HTTP port

You can change the default HTTP port by defining a `server.port` property in your `application.properties` file or by defining an environment variable called `SERVER_PORT`.

You can disable HTTP by setting this variable to `-1` or launch it on a random port by setting it to `0`. This is very handy for testing.

The SSL configuration

Configuring SSL is such a chore, but spring boot has a simple solution. You need only a handful of properties to secure your server:

```
server.port = 8443
server.ssl.key-store = classpath:keystore.jks
server.ssl.key-store-password = secret
server.ssl.key-password = another-secret
```

You will have to generate a keystore file for the above example to work, thought.

We'll have a deeper look at our security options in *Chapter 6, Securing Your Application*. Of course, you can customize the `TomcatEmbeddedServletContainerFactory` function further by adding your own `EmbeddedServletContainerFactory`. This can come in handy if you wish to add multiple connectors, for instance. Refer to the documentation at <http://docs.spring.io/spring-boot/docs/current/reference/html/howto-embedded-servlet-containers.html#howto-configure-ssl> for more information.

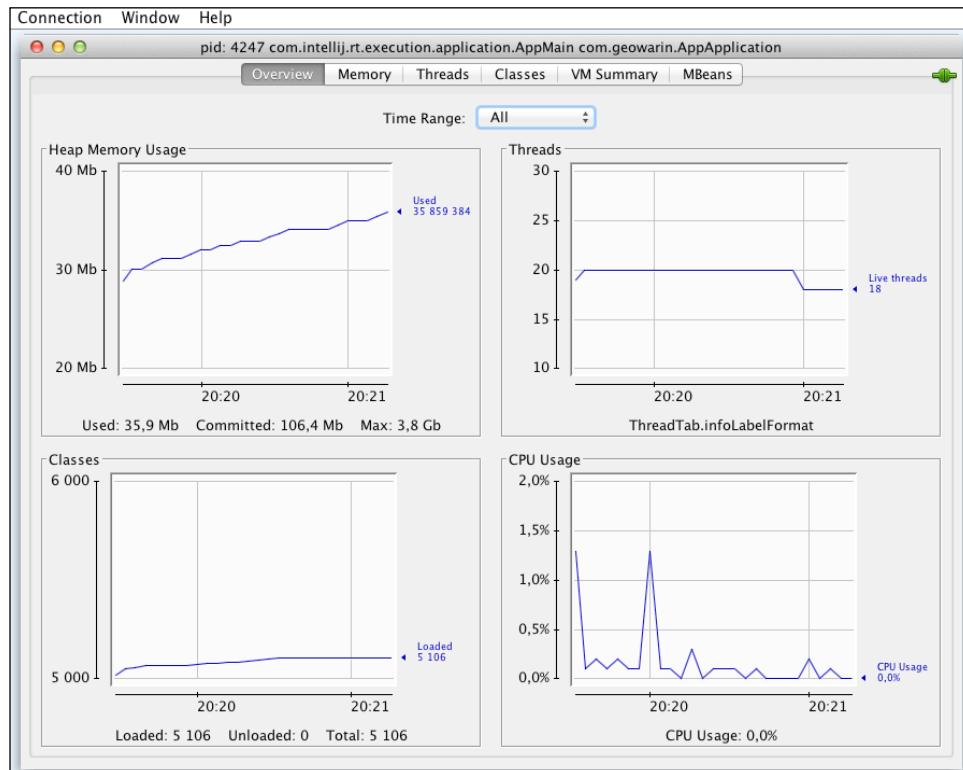
Other configurations

You can add classic Java web elements such as `Servlet`, `Filter`, and `ServletContextListener` by simply declaring them as the `@Bean` elements in your configuration.

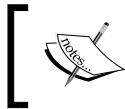
Out of the box, Spring Boot also added three other things for us:

- JSON serialization with Jackson in `JacksonAutoConfiguration`
- Default `HttpMessageConverters` in `HttpMessageConvertersAutoConfiguration`
- JMX capabilities in `JmxAutoConfiguration`

We will see a bit more about the Jackson configuration in *Chapter 5, Crafting a RESTful Application*. About JMX configuration, you can try it out by connecting to your application with `jconsole` locally:



You can add more interesting MBeans by adding `org.springframework.boot:spring-boot-starter-actuator` to your classpath. You can even define your own MBeans and expose them on HTTP using Jolokia. On the other hand, you can also disable those endpoints by adding `spring.jmx.enabled=false` to your configuration.



Refer to <http://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-jmx.html> for more details.

Summary

We now have a very humble spring web application with a RESTful JSON "Hello world" despite having configured nothing ourselves. We have seen what spring boot does for us, how it does it, and hopefully we've got a good idea of how to override the default autoconfiguration.

Detailing how spring boot works is the topic of a book all by itself. If you want to dig deeper, I recommend that you read the excellent book *Learning Spring Boot* by Greg Turnquist in the same collection.

We are now ready for the next chapter where our application will reach a new stage by actually serving web pages, and you will learn more about spring MVC's philosophy.

2

Mastering the MVC Architecture

In this chapter, we will discuss the MVC architecture principles and see how Spring MVC implements those.

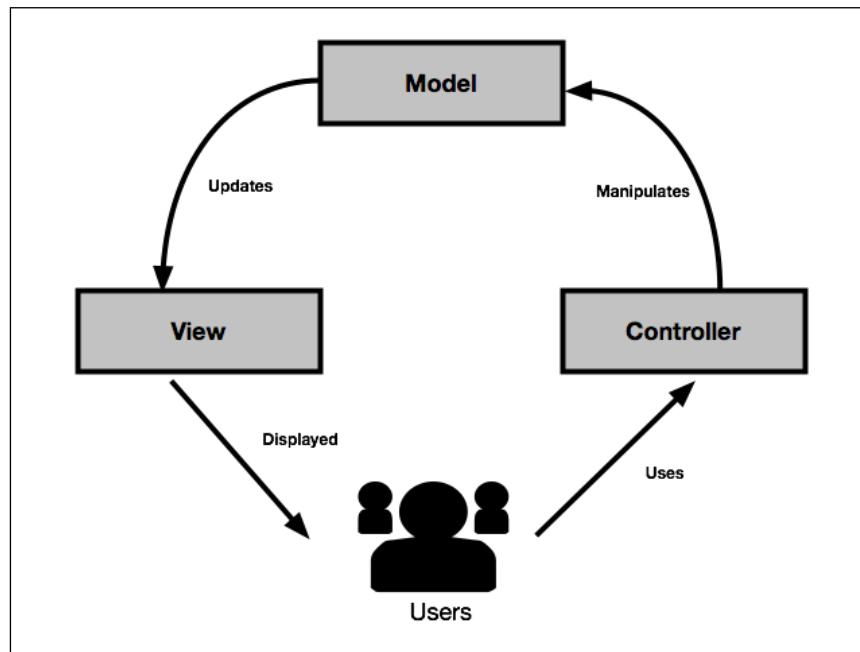
We will continue to use the application from the previous chapter and build something more interesting. Our goal is to design a simple page where users can search for tweets corresponding to certain criteria and display them to our users.

To achieve this, we will use the Spring Social Twitter project, which is available at <http://projects.spring.io/spring-social-twitter/>.

We will see how to make Spring MVC work with a modern template engine, Thymeleaf, and try to understand the inner mechanics of the framework. We will route our users through different views, and finally, we will give a stellar look to our application using WebJars and Materialize (<http://materializecss.com>).

The MVC architecture

I expect the meaning of the MVC acronym to be familiar to most. It stands for Model View Controller, and it is considered to be a very popular way to build a user interface by decoupling the data and the presentation layers.



The MVC pattern became wildly popular after emerging from the world of Smalltalk and landing in the Ruby on Rails framework.

The architectural pattern features three layers:

- **The Model:** This consists of various representations of the data your application knows about.
- **The View:** This is made up of several representations of the data that will be displayed to your users.
- **The Controller:** This is the part of the application that will handle user interactions. It's a bridge between the model and the view.

The idea behind MVC is to decouple the View from the Model. The model must be self-contained and ignorant of the UI. This basically allows the same data to be reused across multiple views. These views are different way to look at the data. Drill down or using different renderers (HTML, PDF) are good illustrations of this principle.

The Controller acts as a mediator between the user and the data. Its role is to control actions available to the end user, as well as routing through the different views of the application.

MVC critics and best practices

While MVC remains the go-to approach for designing a UI, many criticisms arose with its prevalence. Most critics are actually pointing a finger at the incorrect use of the pattern.

Anemic Domain Model

Eric Evans' influential book *Domain Driven Design*, also abbreviated as **DDD**, defines a set of architecture rules leading to a better integration of the business domain inside the code.

One of the core ideas is to take advantage of the object-oriented paradigms inside the domain objects. Going against this principle is sometimes referred to as **Anemic Domain Model**. A good definition of this problem can be found on Martin Fowler's blog (<http://www.martinfowler.com/bliki/AnemicDomainModel.html>).

An Anemic Model typically exhibits the following symptoms:

- The model is constituted from very simple **plain old Java objects (POJOs)** with only getters and setters
- All the business logic is handled inside a service layer
- Validation of the model is found outside this model, for instance, in controllers

This can be a bad practice depending on the complexity of your business domain. Generally speaking, DDD practices require additional efforts to isolate the domain from the application logic.

Architecture is always a tradeoff. It is good to note that typical ways of designing a Spring application can lead to complicated maintenance somewhere along the road.

How to avoid domain anemia is explained here:

- The Service layer is good for application-level abstraction like transaction handling, not business logic.
- Your domain should always be in a valid state. Leave validation inside the form objects using validators or JSR-303's validation annotations.
- translate the inputs into meaningful domain objects.
- Think of your data layer in term of repositories with domain queries (refer to Spring Data Specification, for example)
- Decouple your domain logic from the underlying persistence framework
- Use real objects as much as possible. For instance, manipulate the `FirstName` class rather than a string.

There is much more to DDD than these simple rules: Entities, value types, Ubiquitous Language, Bounded Context, Onion Architecture, and anti corruption layers. I strongly encourage you to study these principles on your own. As far as we are concerned, with this book we will try to keep in mind the guidelines listed earlier as we craft our web application. These concerns will become more familiar to you as we advance through this book.

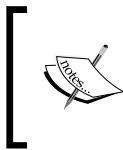
Learning from the sources

If you're familiar with Spring, you have probably already landed on Spring's website, <http://spring.io>. It is entirely made with Spring and the good news is that it is open source.

The code name of the project is sagan. It has numerous interesting features:

- A gradle multimodule project
- Security integration
- Github integration
- Elasticsearch integration
- A JavaScript frontend application

The GitHub wiki associated with the project is really detailed and will help you get started easily with the project.



Visit the following URL if you're interested in the Spring's architecture of a real world application:

<https://github.com/spring-io/sagan>

Spring MVC 1-0-1

In spring MVC, the model is a simple map encapsulated in the `Model` or `ModelAndView` classes of Spring MVC. It can come from a database, files, external services, and so on. It is up to you to define how to fetch the data and put it into the model. The recommended way of interacting with the data layer is through Spring Data libraries: Spring Data JPA, Spring Data MongoDB, and so on. There are a dozen projects related to Spring Data and I encourage you to take a look at <http://projects.spring.io/spring-data>.

The controller side of Spring MVC is handled through the use of the `@Controller` annotation. In a web application, the controller's role is to respond to HTTP requests. Classes annotated with the `@Controller` annotation will be picked up by Spring and given a chance to handle upcoming requests.

Via the `@RequestMapping` annotation, Controllers declare handling specific requests based on their HTTP method (GET or POST methods, for instance) and their URLs. The Controller then decides to either write content directly in the web response or route the application to a view and inject properties into that view.

A pure RESTful application would choose the first approach and expose a JSON or XML representation of the model directly in the HTTP response with the `@ResponseBody` annotation. In the case of a web application, this type of architecture is often associated with a frontend JavaScript framework such as Backbone.js, AngularJS, or React. In this case, the Spring application would then only handle the Model layer of the MVC model. We will study this kind of architecture in *Chapter 4, File Upload and Error Handling*.

With the second approach, the Model is passed to the View, which is rendered by a templating engine and then written to the response.

The view is often associated with a templating dialect, which will allow navigation inside the model. Popular dialects for templating are JSPs, FreeMarker, or Thymeleaf.

Hybrid approaches may take advantage of the templating engine to interact with some aspects of the application and then delegate the view layer to a frontend framework.

Using Thymeleaf

Thymeleaf is a templating engine that gets particular attention from the Spring community.

Its success is due mostly to its friendly syntax (it almost looks like HTML) and the ease with which it can be extended.

Various extensions are available and integrated with Spring Boot:

Support	Dependency
Layouts	nz.net.ultraq.thymeleaf:thymeleaf-layout-dialect
HTML5 data-* attributes	com.github.mxab.thymeleaf.extras:thymeleaf-extras-data-attribute
Internet Explorer conditional comments	org.thymeleaf.extras:thymeleaf-extras-conditionalcomments
Support for spring security	org.thymeleaf.extras:thymeleaf-extras-springsecurity3

A very good tutorial on Thymeleaf's integration with Spring can be found at <http://www.thymeleaf.org/doc/tutorials/2.1/thymeleafspring.html>.

Without further ado, let's add the `spring-boot-starter-thymeleaf` dependency to bootstrap the thymeleaf templating engine:

```
buildscript {
    ext {
        springBootVersion = '1.2.5.RELEASE'
    }
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-
plugin:${springBootVersion}")
        classpath("io.spring.gradle:dependency-management-
plugin:0.5.1.RELEASE")
    }
}
```

```
apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'idea'
apply plugin: 'spring-boot'
apply plugin: 'io.spring.dependency-management'

jar {
    baseName = 'masterSpringMvc'
    version = '0.0.1-SNAPSHOT'
}
sourceCompatibility = 1.8
targetCompatibility = 1.8

repositories {
    mavenCentral()
}

dependencies {
    compile 'org.springframework.boot:spring-boot-starter-web'
    compile 'org.springframework.boot:spring-boot-starter-thymeleaf'
    testCompile 'org.springframework.boot:spring-boot-starter-test'
}

eclipse {
    classpath {
        containers.remove('org.eclipse.jdt.launching.JRE_CONTAINER')
        containers 'org.eclipse.jdt.launching.JRE_CONTAINER/org.
eclipse.jdt.internal.debug.ui.launcher.StandardVMType/JavaSE-1.8'
    }
}

task wrapper(type: Wrapper) {
    gradleVersion = '2.3'
}
```

Our first page

We will now add the first page to our application. It will be located in `src/main/resources/templates`. Let's call the file `resultPage.html`:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head lang="en">
    <meta charset="UTF-8"/>
    <title>Hello thymeleaf</title>
</head>
<body>
    <span th:text="|Hello thymeleaf| " >Hello html</span>
</body>
</html>
```

We can see from the very start that Thymeleaf integrates perfectly with html and its syntax almost feels natural.

The `th:text` value is put between pipes. It means that all the values inside the text will be concatenated.

It might seem a bit awkward at first, but in practice, text will rarely be hardcoded in our pages; so, Thymeleaf makes an opinionated design decision here.

Thymeleaf has a big advantage for web designers: everything that is dynamic inside the templates can fall back to a default value in the case where they are opened without the server running. Resource URLs can be specified relatively and every markup can contain placeholders. In our previous example, the text "Hello html" would not be displayed when the view is rendered in the context of our application, but it will if the file is opened directly with a web browser.

To speed up development, add this property to your `application.properties` file:

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

This will disable the view cache and cause templates to reload every time they are accessed.

Of course, this setting will need to be disabled when we go into production. We will see that in *Chapter 8, Optimizing Your Requests*.

 **Reloading the views**

With the cache disabled, simply save your view with eclipse or use the Build > Make Project action in IntelliJ to refresh the views after a change.

Lastly, we will need to modify our `HelloController` class. Instead of displaying plain text, it must now route to our freshly created view. To accomplish this, we will remove the `@ResponseBody` annotation. Doing so and still returning a string will tell Spring MVC to map this string to a view name instead of displaying a particular model directly in the response.

Here is what our controller now looks like:

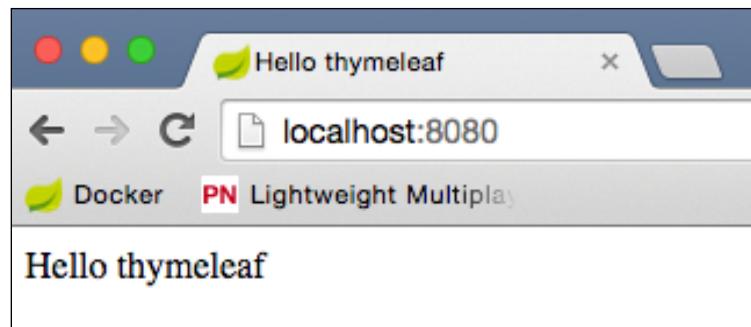
```
@Controller
public class HelloController {

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

In this example, the controller will redirect the user to the view name `resultPage`. The `ViewResolver` interface will then associate this name with our page.

Let's launch our application again and go to `http://localhost:8080`.

You will see the following page:

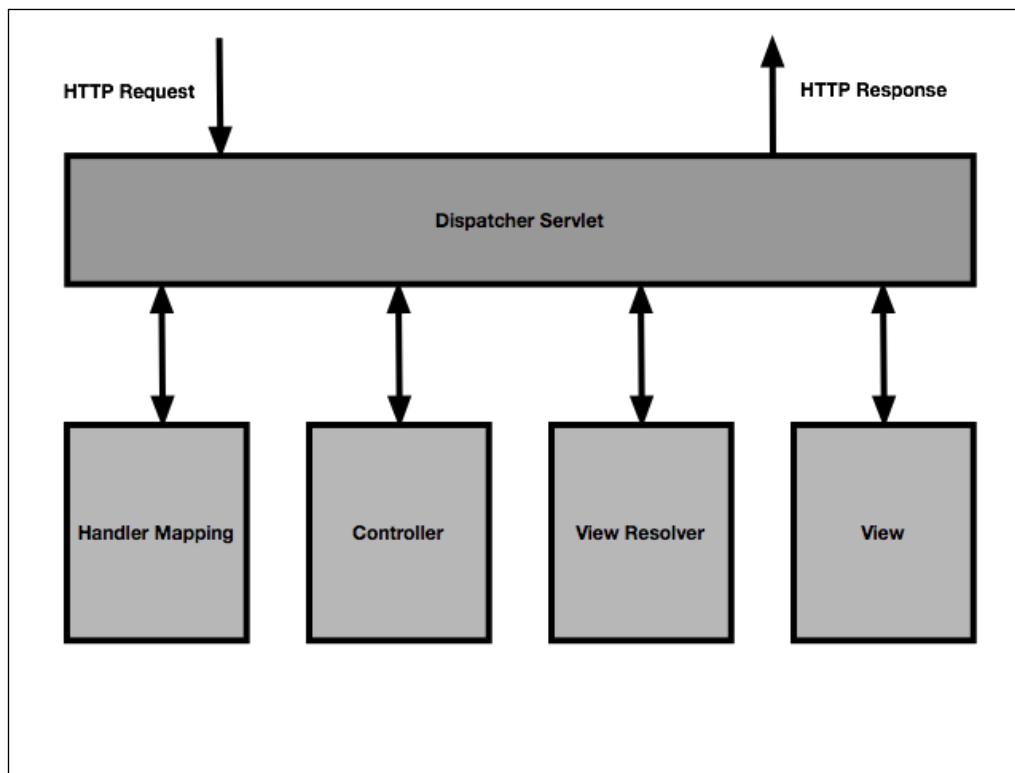


Spring MVC architecture

Let's take a step back from this spectacular new "Hello World" and try to understand what happened inside our web application. To do this, we will retrace the journey of the HTTP request our browser sent and the response it got from the server.

DispatcherServlet

The entry point of every Spring web application is the `DispatcherServlet`. The following figure illustrates the Dispatcher Servlet architecture:



This is a classical `HttpServlet` class that dispatches HTTP requests to `HandlerMapping`. A `HandlerMapping` is an association of resources (URLs) and `Controllers`.

The appropriate methods—those annotated with `@RequestMapping` annotation—are then called on the Controller. In this method, the controller sets the model data and returns the view name to the dispatcher.

The `DispatcherServlet` will then interrogate the `ViewResolver` interface to find the corresponding implementation of the view.

In our case, the `ThymeleafAutoConfiguration` class has set up the view resolver for us.

You can see in the `ThymeleafProperties` class that the default prefix for our views is `classpath:/templates/` the default suffix is `.html`.

This means that, given the view name `resultPage`, the view resolver will look in the `templates` directory of our classpath, looking for a file called `resultPage.html`.

In our application our `ViewResolver` interface is static, but more advanced implementation can return different results given the request headers or the user's locale.

The view will finally be rendered and the result written to the response.

Passing data to the view

Our first page is completely static; it does not really take advantage of the power of Spring MVC. Let's spice things up a little bit. What if the "Hello World" string, instead of being hardcoded, came from the server?

It would still be a lame "hello world" you say? Yes, but it will open up many more possibilities. Let's change our `resultPage.html` file to display a message coming from the model:

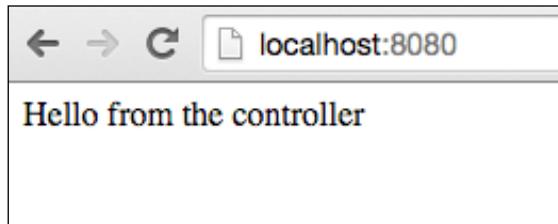
```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head lang="en">
    <meta charset="UTF-8"/>
    <title>Hello thymeleaf</title>
</head>
<body>
    <span th:text="${message}">Hello html</span>
</body>
</html>
```

Then, let's modify our controller so it puts this message inside this model:

```
@Controller
public class HelloController {

    @RequestMapping("/")
    public String hello(Model model) {
        model.addAttribute("message", "Hello from the controller");
        return "resultPage";
    }
}
```

I know, the suspense is killing you! Let's see what `http://localhost:8080` looks like.



The first thing to note is that we passed a new argument to the controller's method and that the `DispatcherServlet` provided the correct object for us. There are, in fact, many objects that can be injected into the controller's methods such as `HttpServletRequest` or `HttpServletResponse`, the `Locale`, the `TimeZone`, and the `Principal`, which represent an authenticated user. The full list of such objects is available in the documentation, which can be found at <http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html#mvc-ann-arguments>.

Spring Expression Language

When using the `{} ${}` syntax, you are in fact using **Spring Expression Language (SpEL)**. There are several variants of EL available in the wild; SpEl is one of the most powerful variants.

Here is an overview of its main features:

Feature	Syntax	Explanation
Accessing a list element	<code>list[0]</code>	
Accessing a map entry	<code>map[key]</code>	
Ternary operator	<code>condition ? 'yes' : 'no'</code>	
Elvis operator	<code>person ?: default</code>	Returns <code>default</code> if <code>person</code> 's value is <code>null</code>
Safe navigation	<code>person?.name</code>	Returns <code>null</code> if <code>person</code> or her name is <code>null</code>
Templating	<code>'Your name is #{person.name}'</code>	Injects values into a string
Projections	<code> \${persons.! [name]} </code>	Extracts the names of all the persons and puts them into a list
Selection	<code>persons.? [name == 'Bob']</code>	Retrieves the person whose name is Bob inside a list
Function call	<code>person.sayHello()</code>	



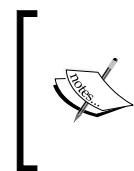
For complete reference, check the manual at <http://docs.spring.io/spring/docs/current/spring-framework-reference/html/expressions.html>.



The SpEl usage is not limited to views. You can also use it in various places inside the Spring framework, for instance, when injecting properties inside beans with the `@Value` annotation.

Getting data with a request parameter

We are able to display data coming from the server inside the view. However, what if we wanted to get input from the user? With the HTTP protocol, there are multiple ways to do this. The simplest way is to pass a query parameter to our URL.



Query parameters

You certainly know query parameters. They are found after the `?` character in a URL. They consist of a list of name and values separated by the `&` symbol (Ampersand), for example, `page?var1=value1&var2=value2`.

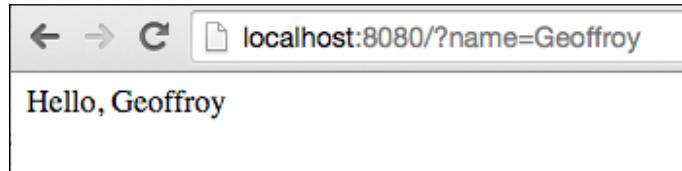


We can leverage this technique to ask our user for their name. Let's modify our `HelloController` class again:

```
@Controller
public class HelloController {

    @RequestMapping("/")
    public String hello(@RequestParam("name") String userName, Model
model) {
        model.addAttribute("message", "Hello, " + userName);
        return "resultPage";
    }
}
```

If we navigate to `localhost:8080/?name=Geoffroy`, we can see the following:



By default, the `request` parameter is mandatory. This means that if we were to navigate to `localhost:8080`, we would see an error message.

Looking at the `@RequestParam` code, we can see that in addition to the `value` parameter, there are two other attributes possible: `required` and `defaultValue`.

Therefore, we can change our code and specify a default value for our parameter or indicate that it is not required:

```
@Controller
public class HelloController {

    @RequestMapping("/")
    public String hello(@RequestParam(defaultValue = "world") String
name, Model model) {
        model.addAttribute("message", "Hello, " + name);
        return "resultPage";
    }
}
```



In Java 8, it is possible not to specify the `value` parameter. In that case, the name of the annotated method parameter will be used.

Enough Hello Worlds, let's fetch tweets!

All right, the name of the book isn't "Mastering Hello Worlds", after all. With Spring, interrogating Twitter's API is really easy.

Registering your application

Before you start, you have to register your application in the Twitter developer console.

Go to <https://apps.twitter.com> and create a new application.

Give it the name you please. Under the website and Callback URL sections, just enter `http://127.0.0.1:8080`. This will allow you to test your application in development on your local machine.

Create an application

Application Details

Name *
MasteringSpringMVC4
Your application name. This is used to attribute the source of a tweet and in user-facing authorization screens.

Description *
An application to master Spring MVC4
Your application description, which will be shown in user-facing authorization screens. Between 10 and 64 characters.

Website *
`http://127.0.0.1:8080`
Your application's publicly accessible home page, where users can go to download, make use of, or source attribution for tweets created by your application and will be shown in user-facing authorization screens (if you don't have a URL yet, just put a placeholder here but remember to change it later.)

Callback URL
`http://127.0.0.1:8080`
Where should we return after successfully authenticating? OAuth 1.0a applications should explicitly give a URL here. To restrict your application from using callbacks, leave this field blank.

Now, navigate to the keys, access the token, and copy the **Consumer Key** and the **Consumer Secret**. We will use this in a moment. Take a look at the following screenshot:

Details Settings Keys and Access Tokens Permissions

Application Settings

Keep the "Consumer Secret" a secret. This key should never be human-readable in your application.

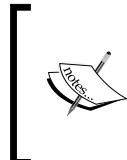
Consumer Key (API Key)	[REDACTED]
Consumer Secret (API Secret)	[REDACTED]
Access Level	Read-only (modify app permissions)
Owner	geowarin
Owner ID	135249820

By default, our application has read only permissions. This will be enough for our application, but you can tweak it if you wish.

Setting up Spring Social Twitter

We will add the following dependency to our `build.gradle` file:

```
compile 'org.springframework.boot:spring-boot-starter-social-twitter'
```



Spring Social is a set of projects providing access to the public APIs of various social networks. Out of the box, Spring Boot provides integration with Twitter, Facebook, and LinkedIn. Spring Social includes about 30 projects overall, which can be found at <http://projects.spring.io/spring-social/>.

Add the following two lines to the `application.properties`:

```
spring.social.twitter.appId= <Consumer Key>
spring.social.twitter.appSecret= <Consumer Secret>
```

These are the keys associated with the application we just created.

You will learn more about OAuth in *Chapter 6, Securing Your Application*. For now, we will just use those credentials to issue requests to Twitter's API on behalf of our application.

Accessing Twitter

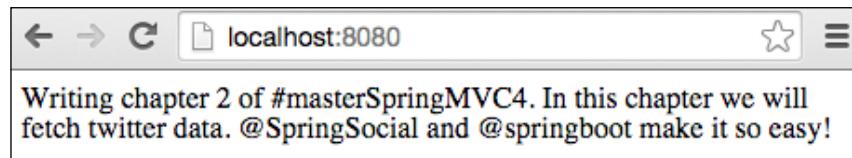
We can now use Twitter in our controller. Let's change its name to TweetController as a variable to reflect its new responsibility in a better manner:

```
@Controller
public class HelloController {

    @Autowired
    private Twitter twitter;

    @RequestMapping("/")
    public String hello(@RequestParam(defaultValue =
"masterSpringMVC4") String search, Model model) {
        SearchResults searchResults = twitter.searchOperations().search(search);
        String text = searchResults.getTweets().get(0).getText();
        model.addAttribute("message", text);
        return "resultPage";
    }
}
```

As you can see, the code searches for tweets matching the request parameter. If it all goes well, you will see the text of the first one being displayed on your screen:



Of course, if the search doesn't yield any result, our clumsy code will fail with an `ArrayOutOfBoundsException`. So, do not hesitate to tweet to solve the problem!

What if we wanted to display a list of tweets? Let's modify the `resultPage.html` file:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head lang="en">
    <meta charset="UTF-8"/>
    <title>Hello twitter</title>
```

```
</head>
<body>
    <ul>
        <li th:each="tweet : ${tweets}" th:text="${tweet}">Some
        tweet</li>
    </ul>
</body>
</html>
```

 The `th:each` is a tag defined in Thymeleaf that allows it to iterate over a collection and assign each value to a variable inside a loop.

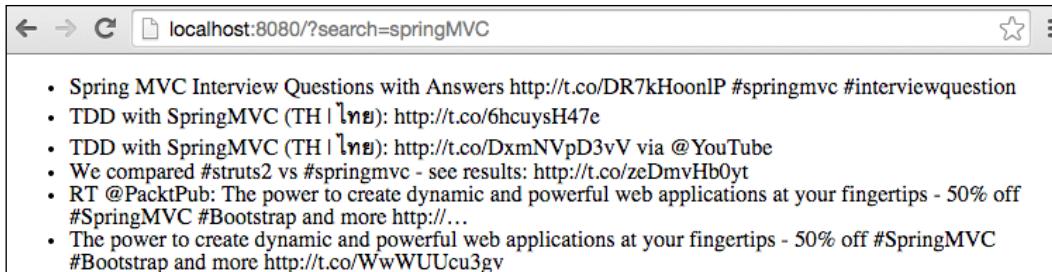
We will need to change our controller as well:

```
@Controller
public class TweetController {

    @Autowired
    private Twitter twitter;

    @RequestMapping("/")
    public String hello(@RequestParam(defaultValue =
"masterSpringMVC4") String search, Model model) {
        SearchResults searchResults = twitter.searchOperations().search(search);
        List<String> tweets =
            searchResults.getTweets()
                .stream()
                .map(Tweet::getText)
                .collect(Collectors.toList());
        model.addAttribute("tweets", tweets);
        return "resultPage";
    }
}
```

Note that we are using Java 8 streams to collect only the messages from the tweets. The Tweet class contains many other attributes such as the sender, the retweet count, and so on. However, we will keep it simple for now, as shown in the following screenshot:



Java 8 streams and lambdas

You might not be familiar with lambdas yet. In Java 8, every collection gets a default method `stream()`, which gives access to functional-style operations.

These operations can be either intermediate operations returning a stream, and thus allowing chaining, or a terminal operation that returns a value.

The most famous intermediate operations are as follows:

- `map`: This applies a method to every element of a list and returns the list of results
- `filter`: This returns a list of every element matching a predicate
- `reduce`: This projects a list into a single value using an operation and an accumulator

Lambdas are shorthand syntax for function expressions. They can be coerced into a Single Abstract Method, an interface with only one function.

For instance, you can implement the `Comparator` interface as follows:

```
Comparator<Integer> c = (e1, e2) -> e1 - e2;
```

Within lambdas, the `return` keyword is implicitly its last expression.

The double colon operator we used earlier is a shortcut to get a reference to a function on a class,

```
Tweet::getText
```

The preceding is equivalent to the following:

```
(Tweet t) -> t.getText()
```

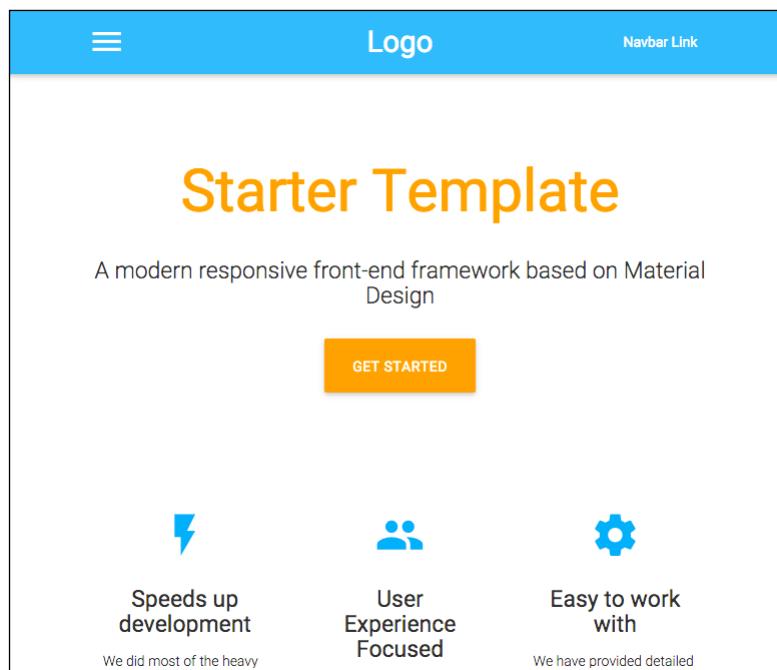
The `collect` method allows us to call a terminal operation. The `Collectors` class is a set of terminal operations that will put results into lists, sets, or maps, allowing grouping, joining, and so on.

Calling the `collect(Collectors.toList())` method will produce a list with every element within the stream; in our case, the tweet names.

Material design with WebJars

Our application is already great but it seriously leaves something to be desired in terms of aesthetics. You may have heard of material design. It is Google's take on flat design.

We will use Materialize (<http://materializecss.com>), a great looking responsive CSS and JavaScript library, just like Bootstrap.



We talked a bit about WebJars in *Chapter 1, Setting Up a Spring Web Application in No Time*; we will now get to use them. Add jQuery and Materialize CSS to our dependencies:

```
compile 'org.webjars:materializecss:0.96.0'  
compile 'org.webjars:jquery:2.1.4'
```

The way a WebJar is organized is completely standardized. You will find the JS and CSS files of any library in `/webjars/{lib}/{version}/*.js`.

For instance, to add jQuery to our page, the following to a web page:

```
<script src="/webjars/jquery/2.1.4/jquery.js"></script>
```

Let's modify our controller so that it gives us a list of all tweet objects instead of simple text:

```
package masterSpringMvc.controller;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.social.twitter.api.SearchResults;  
import org.springframework.social.twitter.api.Tweet;  
import org.springframework.social.twitter.api.Twitter;  
import org.springframework.stereotype.Controller;  
import org.springframework.ui.Model;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RequestParam;  
  
import java.util.List;  
  
@Controller  
public class TweetController {  
  
    @Autowired  
    private Twitter twitter;  
  
    @RequestMapping("/")  
    public String hello(@RequestParam(defaultValue =  
"masterSpringMVC4") String search, Model model) {  
        SearchResults searchResults = twitter.searchOperations().  
        search(search);  
        List<Tweet> tweets = searchResults.getTweets();  
        model.addAttribute("tweets", tweets);  
        model.addAttribute("search", search);  
        return "resultPage";  
    }  
}
```

Let's include materialize CSS in our view:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head lang="en">
    <meta charset="UTF-8"/>
    <title>Hello twitter</title>

    <link href="/webjars/materializecss/0.96.0/css/materialize.css"
type="text/css" rel="stylesheet" media="screen,projection"/>
</head>
<body>
<div class="row">

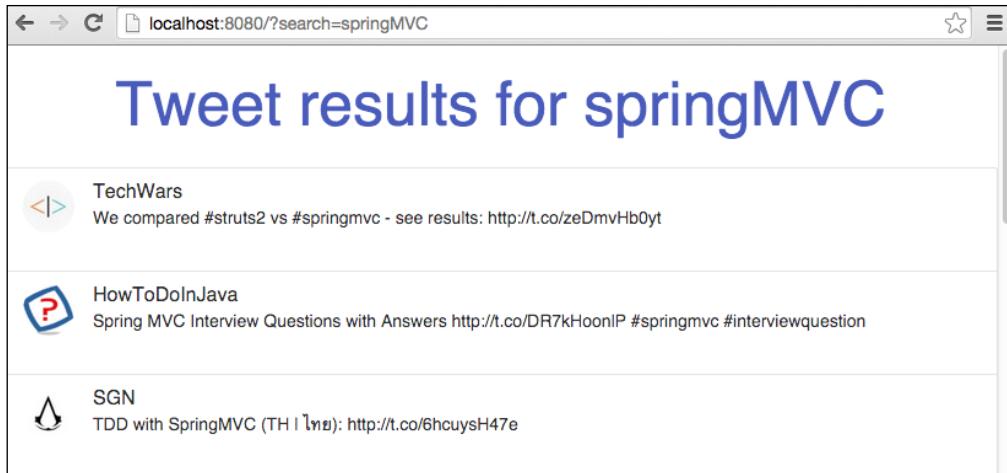
    <h2 class="indigo-text center" th:text="|Tweet results for
${search} | ">Tweets</h2>

    <ul class="collection">
        <li class="collection-item avatar" th:each="tweet :
${tweets}">
            
            <span class="title" th:text="${tweet.user.
name}">Username</span>
            <p th:text="${tweet.text}">Tweet message</p>
        </li>
    </ul>

</div>

<script src="/webjars/jquery/2.1.4/jquery.js"></script>
<script src="/webjars/materializecss/0.96.0/js/materialize.js"></
script>
</body>
</html>
```

The result already looks way better!



Using layouts

The last thing we want to do is to put the reusable chunks of our UI into templates. To do this, we will use the `thymeleaf-layout-dialect` dependency, which is included in the `spring-boot-starter-thymeleaf` dependency of our project.

We will create a new file called `default.html` in `src/main/resources/templates/layout`. It will contain the code we will repeat from page to page:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout">
<head>
    <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8" />
    <meta name="viewport" content="width=device-width, initial-
scale=1, maximum-scale=1.0, user-scalable=no" />
    <title>Default title</title>

    <link href="/webjars/materializecss/0.96.0/css/materialize.css"
          type="text/css" rel="stylesheet" media="screen,projection"/>
</head>
<body>
```

```
<section layout:fragment="content">
    <p>Page content goes here</p>
</section>

<script src="/webjars/jquery/2.1.4/jquery.js"></script>
<script src="/webjars/materializecss/0.96.0/js/materialize.js"></script>
</body>
</html>
```

We will now modify the `resultPage.html` file so it uses the layout, which will simplify its contents:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      layout:decorator="layout/default">
<head lang="en">
    <title>Hello twitter</title>
</head>
<body>
<div class="row" layout:fragment="content">

    <h2 class="indigo-text center" th:text="|Tweet results for
${search}|">Tweets</h2>

    <ul class="collection">
        <li class="collection-item avatar" th:each="tweet :
${tweets}">
            
            <span class="title" th:text="${tweet.user.
name}">Username</span>

            <p th:text="${tweet.text}">Tweet message</p>
        </li>
    </ul>
</div>
</body>
</html>
```

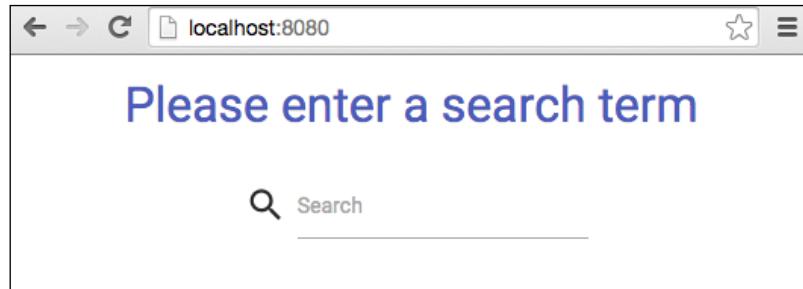
The `layout:decorator="layout/default"` will indicate where our layout can be found. We can then inject content into the different `layout:fragment` sections of the layout. Note that each template are valid HTML files. You can also override the title very easily.

Navigation

We have a nice little tweet display application, but how are our users supposed to figure out that they need to supply a "search" request parameter?

It would be nice if we added a little form to our application.

Let's do something like this:



First, we need to modify our TweetController to add a second view to our application. The search page will be available directly at the root of our application and the result page when hit enter in the search field:

```
@Controller
public class TweetController {

    @Autowired
    private Twitter twitter;

    @RequestMapping("/")
    public String home() {
        return "searchPage";
    }

    @RequestMapping("/result")
    public String hello(@RequestParam(defaultValue =
"masterSpringMVC4") String search, Model model) {
        SearchResults searchResults = twitter.searchOperations().search(search);
        List<Tweet> tweets = searchResults.getTweets();
        model.addAttribute("tweets", tweets);
        model.addAttribute("search", search);
        return "resultPage";
    }
}
```

We will add another page to the `templates` folder called the `searchPage.html` file. It will contain a simple form, which will pass the search term to the result page via the `get` method:

```
<!DOCTYPE html>
<html xmlns:th="http://www.w3.org/1999/xhtml"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      layout:decorator="layout/default">
<head lang="en">
    <title>Search</title>
</head>
<body>

    <div class="row" layout:fragment="content">

        <h4 class="indigo-text center">Please enter a search term</h4>

        <form action="/result" method="get" class="col s12">
            <div class="row center">
                <div class="input-field col s6 offset-s3">
                    <i class="mdi-action-search prefix"></i>
                    <input id="search" name="search" type="text" class="validate"/>
                    <label for="search">Search</label>
                </div>
            </div>
        </form>
    </div>

    </body>
</html>
```

This is very simple HTML and it works perfectly. You can try it now.

What if we wanted to disallow some search result? Let's say we want to display an error message if the user types in `struts`.

The best way to achieve this would be to modify the form to post the data. In the controller, we can then intercept what is posted and implement this business rule accordingly.

First, we need to change the form in the `searchPage`, which is as follows:

```
<form action="/result" method="get" class="col s12">
```

Now, we change the form to this:

```
<form action="/postSearch" method="post" class="col s12">
```

We also need to handle this post on the server. Add this method to the TweetController:

```
@RequestMapping(value = "/postSearch", method = RequestMethod.POST)
public String postSearch(HttpServletRequest request,
    RedirectAttributes redirectAttributes) {
    String search = request.getParameter("search");
    redirectAttributes.addAttribute("search", search);
    return "redirect:result";
}
```

There are several novelties here:

- In the request mapping annotation, we specify the HTTP method we want to handle, that is, POST.
- We inject two attributes directly as method parameters. They are the request and RedirectAttributes.
- We retrieve the value posted on the request and pass it on to the next view.
- Instead of returning the name of the view, we make a redirection to a URL.

The RedirectAttributes is a Spring model that will be specifically used to propagate values in a redirect scenario.

 **Redirect/Forward** are classical options in the context of a Java web application. They both change the view that is displayed on the user's browser. The difference is that **Redirect** will send a 302 header that will trigger navigation inside the browser, whereas **Forward** will not cause the URL to change. In Spring MVC, you can use either option simply by prefixing your method return strings with `redirect:` or `forward:`. In both cases, the string you return will not be resolved to a view like we saw earlier, but will instead trigger navigation to a specific URL.

The preceding example is a bit contrived, and we will see smarter form handling in the next chapter. If you put a breakpoint in the `postSearch` method, you will see that it will be called right after a post in our form.

So what about the error message?

Let's change the `postSearch` method:

```
@RequestMapping(value = "/postSearch", method = RequestMethod.POST)
public String postSearch(HttpServletRequest request,
    RedirectAttributes redirectAttributes) {
    String search = request.getParameter("search");
    if (search.toLowerCase().contains("struts")) {
        redirectAttributes.addFlashAttribute("error", "Try
using spring instead!");
        return "redirect:/";
    }
    redirectAttributes.addAttribute("search", search);
    return "redirect:result";
}
```

If the user's search terms contain "struts", we redirect them to the `searchPage` and add a little error message using flash attributes.

These special kinds of attributes live only for the time of a request and will disappear when the page is refreshed. This is very useful when we use the POST-REDIRECT-GET pattern, as we just did.

We will need to display this message in the `searchPage` result:

```
<!DOCTYPE html>
<html xmlns:th="http://www.w3.org/1999/xhtml"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      layout:decorator="layout/default">
<head lang="en">
    <title>Search</title>
</head>
<body>

<div class="row" layout:fragment="content">

    <h4 class="indigo-text center">Please enter a search term</h4>

    <div class="col s6 offset-s3">
        <div id="errorMessage" class="card-panel red lighten-2"
th:if="${error}">
            <span class="card-title" th:text="${error}"></span>
        </div>
    </div>
</div>
```

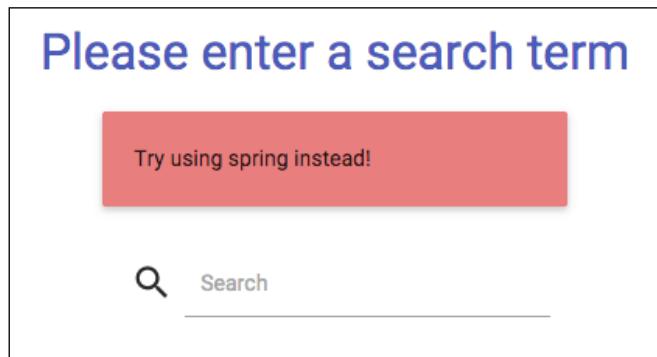
```

<form action="/postSearch" method="post" class="col s12">
    <div class="row center">
        <div class="input-field">
            <i class="mdi-action-search prefix"></i>
            <input id="search" name="search" type="text" class="validate"/>
            <label for="search">Search</label>
        </div>
    </div>
</form>
</div>
</div>

</body>
</html>

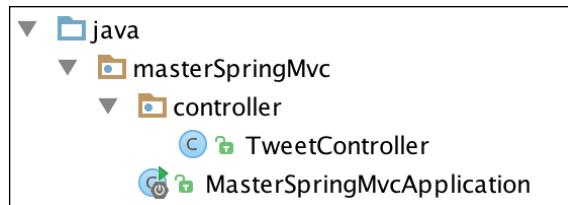
```

Now, if users try to search for "struts2" tweets, they will get a useful and appropriate answer:



The check point

At the end of this chapter, you should have one controller, the `TweetController`, handling the search and the untouched generated configuration class, `MasterSpringMvcApplication`, in the `src/main/java` directory:



In the `src/main/resources` directory, you should have one default layout and two pages using it.

In the `application.properties` file, we added the Twitter application credentials as well as a property telling Spring not to cache the templates to ease development:



Summary

In this chapter, you learned what it takes to make a good MVC architecture. We saw some of the inner workings of Spring MVC and used Spring Social Twitter with very little configuration. We can now design a beautiful web application, thanks to WebJars.

In the next chapter, we will ask the user to fill in their profile, so that we can fetch tweets they might like automatically. This will give you the opportunity to learn more about forms, formatting, validation, and internationalization.

3

Handling Forms and Complex URL Mapping

Our application, as beautiful as it looks, would benefit from more informations about our users.

We could ask them to provide the topics they are interested in.

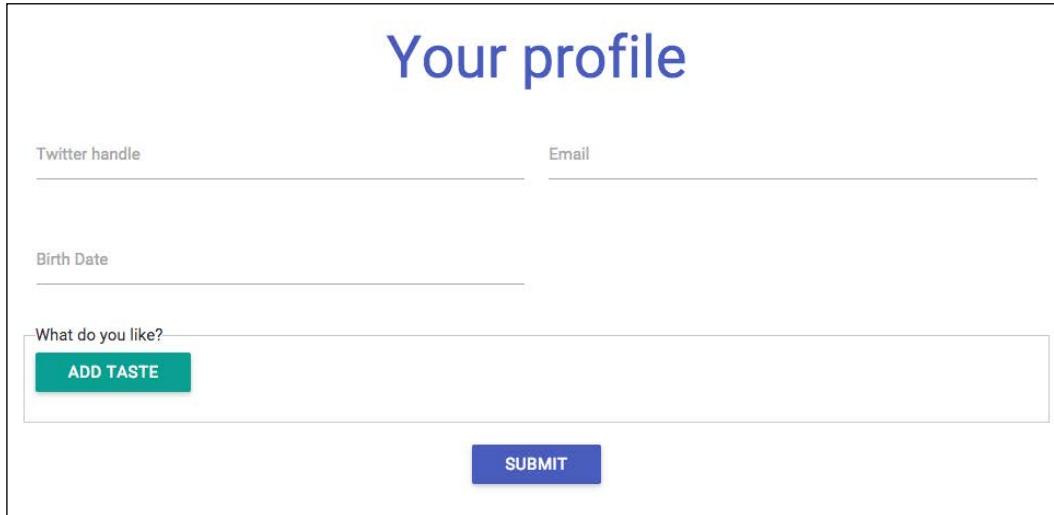
In this chapter, we will build a profile page. It will feature server- and client-side validation and file upload for a profile picture. We will save that information in the user session and also ensure that our audience is as large as possible by translating the application into several languages. Finally, we will display a summary of Twitter activity matching users' tastes.

Sounds good? Let's get started, we have some work to do.

The profile page – a form

Forms are the cornerstones of every web application. They have been the main way to get user input since the very beginning of the Internet!

Our first task here is to create a profile page like this one:



Twitter handle

Email

Birth Date

What do you like?

ADD TASTE

SUBMIT

It will let the user enter some personal information as well as a list of tastes. These tastes will then be fed to our search engine.

Let's create a new page in `templates/profile/profilePage.html`:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      layout:decorator="layout/default">
<head lang="en">
    <title>Your profile</title>
</head>
<body>
<div class="row" layout:fragment="content">

    <h2 class="indigo-text center">Personal info</h2>

    <form th:action="@{/profile}" method="post" class="col m8 s12
offset-m2">

        <div class="row">
            <div class="input-field col s6">
                <input id="twitterHandle" type="text"/>
                <label for="twitterHandle">Last Name</label>
            </div>
    
```

```
<div class="input-field col s6">
    <input id="email" type="text"/>
    <label for="email">Email</label>
</div>
</div>
<div class="row">
    <div class="input-field col s6">
        <input id="birthDate" type="text"/>
        <label for="birthDate">Birth Date</label>
    </div>
</div>
<div class="row s12">
    <button class="btn waves-effect waves-light" type="submit" name="save">Submit
        <i class="mdi-content-send right"></i>
    </button>
</div>
</form>
</div>
</body>
</html>
```

Note the `@{ }` syntax that will construct the full path to a resource by prepending the server context path (in our case, `localhost:8080`) to its argument.

We will also create the associated controller named `ProfileController` in the `profile` package:

```
package masterspringmvc4.profile;

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

@Controller
public class ProfileController {

    @RequestMapping("/profile")
    public String displayProfile() {
        return "profile/profilePage";
    }
}
```

Now, you can go to `http://localhost:8080` and behold a beautiful form that does nothing. That's because we didn't map any action to the post URL.

Let's create a **Data Transfer Object (DTO)** in the same package as our controller. We will name it `ProfileForm`. Its role will be to map the fields of our web form and describe validation rules:

```
package masterSpringMvc.profile;

import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;

public class ProfileForm {
    private String twitterHandle;
    private String email;
    private LocalDate birthDate;
    private List<String> tastes = new ArrayList<>();

    // getters and setters
}
```

This is a regular **Plain Old Java Object (POJO)**. Don't forget to generate the getters and setters, without which our data binding will not work properly. Note that we have a list of tastes that we will not populate right now but a bit later.

Since we are using Java 8, the birth date of our user will be using the new Java date-time API (JSR 310). This API is much better than the old `java.util.Date` API because it makes strong distinctions between all the nuances of human dates and uses a fluent API and immutable data structures.

In our example, a `LocalDate` class is a simple day without time associated to it. It can be differentiated from the `LocalTime` class, which represents a time within a day, the `LocalDateTime` class, which represents both, or the `ZonedDateTime` class, which uses a time zone.



If you wish to learn more about the Java 8 date time API, refer to the Oracle tutorial available at <https://docs.oracle.com/javase/tutorial/datetime/TOC.html>.



Good advice is to always generate the `toString` method of our data objects like this form. It is extremely useful for debugging.

To instruct Spring to bind our field to this DTO, we will have to add some metadata in the `profilePage`:

```

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      layout:decorator="layout/default">
<head lang="en">
    <title>Your profile</title>
</head>
<body>
    <div class="row" layout:fragment="content">

        <h2 class="indigo-text center">Personal info</h2>

        <form th:action="@{/profile}" th:object="${profileForm}"
              method="post" class="col m8 s12 offset-m2">

            <div class="row">
                <div class="input-field col s6">
                    <input th:field="${profileForm.twitterHandle}"
                           id="twitterHandle" type="text"/>
                    <label for="twitterHandle">Last Name</label>
                </div>
                <div class="input-field col s6">
                    <input th:field="${profileForm.email}" id="email"
                           type="text"/>
                    <label for="email">Email</label>
                </div>
            </div>
            <div class="row">
                <div class="input-field col s6">
                    <input th:field="${profileForm.birthDate}"
                           id="birthDate" type="text"/>
                    <label for="birthDate">Birth Date</label>
                </div>
            </div>
            <div class="row s12">
                <button class="btn waves-effect waves-light" type="submit"
                       name="save">Submit
                    <i class="mdi-content-send right"></i>
                </button>
            </div>
        </form>
    </div>
</body>
</html>

```

You will notice two things:

- The `th:object` attribute in the form
- The `th:field` attributes in all the fields

The first one will bind an object by its type to the controller. The second ones will bind the actual fields to our form bean attributes.

For the `th:object` field to work, we need to add an argument of the type `ProfileForm` to our request mapping methods:

```
@Controller
public class ProfileController {

    @RequestMapping("/profile")
    public String displayProfile(ProfileForm profileForm) {
        return "profile/profilePage";
    }

    @RequestMapping(value = "/profile", method = RequestMethod.POST)
    public String saveProfile(ProfileForm profileForm) {
        System.out.println("save ok" + profileForm);
        return "redirect:/profile";
    }
}
```

We also added a mapping for the `POST` method that will be called when the form is submitted. At this point, if you try to submit the form with a date (for instance `10/10/1980`), it won't work at all and give you an error 400 and no useful logging information.

Logging in Spring Boot

With Spring Boot, logging configuration is extremely simple. Just add `logging.level.{package} = DEBUG` to the application properties file, where `{package}` is the fully qualified name of one of the classes or a package in your application. You can, of course, replace `debug` by any logging level you want. You can also add a classic logging configuration. Refer to <http://docs.spring.io/spring-boot/docs/current/reference/html/howto-logging.html> for more information.

We will need to debug our application a little bit to understand what happened. Add this line to your file `application.properties`:

```
logging.level.org.springframework.web=DEBUG
```

The `org.springframework.web` package is the base package of Spring MVC. This will allow us to see debug information generated by Spring web. If you submit the form again, you will see the following error in the log:

```
Field error in object 'profileForm' on field 'birthDate':  
rejected value [10/10/1980]; codes [typeMismatch.profileForm.  
birthDate,typeMismatch.birthDate,typeMismatch.java.time.  
LocalDate,typeMismatch]; ... nested exception is org.springframework.  
core.convert.ConversionFailedException: Failed to convert from type  
java.lang.String to type java.time.LocalDate for value '10/10/1980';  
nested exception is java.time.format.DateTimeParseException: Text  
'10/10/1980' could not be parsed, unparsed text found at index 8]
```

To understand what's going on, we need to have a look at the `DateTimeFormatterRegistrar` class of Spring.

In this class, you will see half a dozen parsers and printers for the JSR 310. They will all fall back on the short style date format, which is either `MM/dd/yy` if you live in the US or `dd/MM/yy` otherwise.

This will instruct Spring Boot to create a `DateTimeFormatter` class when our application starts.

We need to do the same thing in our case and create our own formatter since writing a year with two digits is a bit awkward.

A `Formatter` in Spring is a class that can both print and parse an object. It will be used to decode and print a value from and to a `String`.

We will create a very simple formatter in the `date` package called `USLocalDateFormatter`:

```
public class USLocalDateFormatter implements Formatter<LocalDate> {  
    public static final String US_PATTERN = "MM/dd/yyyy";  
    public static final String NORMAL_PATTERN = "dd/MM/yyyy";  
  
    @Override public LocalDate parse(String text, Locale locale)  
    throws ParseException {  
        return LocalDate.parse(text, DateTimeFormatter.  
        ofPattern(getPattern(locale)));  
    }  
}
```

```
    @Override public String print(LocalDate object, Locale locale) {
        return DateTimeFormatter.ofPattern(getPattern(locale)).
format(object);
    }

    public static String getPattern(Locale locale) {
        return isUnitedStates(locale) ? US_PATTERN : NORMAL_PATTERN;
    }

    private static boolean isUnitedStates(Locale locale) {
        return Locale.US.getCountry().equals(locale.getCountry());
    }
}
```

This little class will allow us to parse the date in a more common format (with years in four digits) according to the user's locale.

Let's create a new class in the `config` package called `WebConfiguration`:

```
package masterSpringMvc.config;

import masterSpringMvc.dates.USLocalDateFormatter;
import org.springframework.context.annotation.Configuration;
import org.springframework.format.FormatterRegistry;
import org.springframework.web.servlet.config.annotation.
WebMvcConfigurerAdapter;

import java.time.LocalDate;

@Configuration
public class WebConfiguration extends WebMvcConfigurerAdapter {

    @Override public void addFormatters(FormatterRegistry registry) {
        registry.addFormatterForFieldType(LocalDate.class, new
USLocalDateFormatter());
    }
}
```

This class extends the `WebMvcConfigurerAdapter`, which is a very handy class to customize the Spring MVC configuration. It provides a lot of common extension points that you can access by overriding methods such as the `addFormatters()` method.

This time, submitting our form won't result in any error except if you don't type the date with the correct date format.

For the moment, it is impossible for the users to see the format in which they are supposed to enter their birth date, so let's add this information to the form.

In the `ProfileController`, let's add a `dateFormat` attribute:

```
@ModelAttribute("dateFormat")
public String localeFormat(Locale locale) {
    return USLocalDateFormatter.getPattern(locale);
}
```

The `@ModelAttribute` annotation will allow us to expose a property to the web page, exactly like the `model.addAttribute()` method that we saw in the previous chapter.

Now, we can use this information in our page by adding a placeholder to our date field:

```
<div class="row">
    <div class="input-field col s6">
        <input th:field="${profileForm.birthDate}" id="birthDate"
type="text" th:placeholder="${dateFormat}"/>
        <label for="birthDate">Birth Date</label>
    </div>
</div>
```

This information will now be displayed to the user:



Validation

We wouldn't want our user to enter invalid or empty information and that's why we will need to add some validation logic to our `ProfileForm`.

```
package masterspringmvc4.profile;

import org.hibernate.validator.constraints.Email;
import org.hibernate.validator.constraints.NotEmpty;
```

```
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Past;
import javax.validation.constraints.Size;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

public class ProfileForm {
    @Size(min = 2)
    private String twitterHandle;

    @Email
    @NotEmpty
    private String email;

    @NotNull
    private Date birthDate;

    @NotEmpty
    private List<String> tastes = new ArrayList<>();
}
```

As you can see, we added a few validation constraints. These annotations come from the JSR-303 specification, which specifies bean validation. The most popular implementation of this specification is `hibernate-validator`, which is included in Spring Boot.

You can see that we use annotations coming from the `javax.validation.constraints` package (defined in the API) and some coming from the `org.hibernate.validator.constraints` package (additional constraints). Both work, I encourage you to take a look at what is available in those packages in the jars `validation-api` and `hibernate-validator`.

You can also take a look at the constraints available in the hibernate validator in the documentation at http://docs.jboss.org/hibernate/stable/validator/reference/en-US/html_single/#section-builtin-constraints.

We will need to add a few more things for validation to work. First, the controller needs to say that it wants a valid model on form submission. Adding the `javax.validation.Valid` annotation to the parameter representing the form does just that:

```
@RequestMapping(value = "/profile", method = RequestMethod.POST)
public String saveProfile(@Valid ProfileForm profileForm,
    BindingResult bindingResult) {
    if (bindingResult.hasErrors()) {
```

```

        return "profile/profilePage";
    }

    System.out.println("save ok" + profileForm);
    return "redirect:/profile";
}

```

Note that we do not redirect the user if the form contains any errors. This will allow us to display them on the same web page.

Speaking of which, we need to add a place on the web page where those errors will be displayed.

Add these lines just at the beginning of the form tag in `profilePage.html`:

```

<ul th:if="${#fields.hasErrors('*')}" class="errorlist">
    <li th:each="err : ${#fields.errors('*')}" th:text="${err}">Input
        is incorrect</li>
</ul>

```

This will iterate through every error found in the form and display them in a list. If you try to submit an empty form, you will see a bunch of errors:

Personal info

may not be empty
 may not be empty
 size must be between 2 and 2147483647
 may not be null

Last Name	Email
<hr/>	<hr/>

Note that the `@NotEmpty` check on the tastes will prevent the form from being submitted. Indeed, we do not yet have a way to provide them.

Customize validation messages

These error messages are not very useful for our user yet. The first thing we need to do is to associate them properly to their respective fields. Let's modify `profilePage.html`:

```

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

```

```
xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
layout:decorator="layout/default">
<head lang="en">
    <title>Your Profile</title>
</head>
<body>
<div class="row" layout:fragment="content">

    <h2 class="indigo-text center">Personal info</h2>

    <form th:action="@{/profile}" th:object="${profileForm}"
method="post" class="col m8 s12 offset-m2">

        <div class="row">
            <div class="input-field col s6">
                <input th:field="${profileForm.twitterHandle}"
id="twitterHandle" type="text" th:errorclass="invalid"/>
                <label for="twitterHandle">Twitter handle</label>

                <div th:errors="*{twitterHandle}" class="red-
text">Error</div>
            </div>
            <div class="input-field col s6">
                <input th:field="${profileForm.email}" id="email"
type="text" th:errorclass="invalid"/>
                <label for="email">Email</label>

                <div th:errors="*{email}" class="red-text">Error</div>
            </div>
        </div>
        <div class="row">
            <div class="input-field col s6">
                <input th:field="${profileForm.birthDate}"
id="birthDate" type="text" th:errorclass="invalid" th:placeholder="${
dateFormat}"/>
                <label for="birthDate">Birth Date</label>

                <div th:errors="*{birthDate}" class="red-text">Error</
div>
            </div>
        </div>
        <div class="row s12">
            <button class="btn indigo waves-effect waves-light"
type="submit" name="save">Submit
            <i class="mdi-content-send right"></i>
        </div>
    </form>
</div>
```

```

        </button>
    </div>
</form>
</div>
</body>
</html>

```

You will notice that we added a `th:errors` tag below each field in the form. We also added a `th:errorclass` tag to each field. If the field contains an error, the associated css class will be added to the DOM.

The validation looks much better already:

The screenshot shows a form titled 'Personal info'. It has two fields: 'Twitter handle' and 'Email'. The 'Twitter handle' field has a red border and the error message 'size must be between 2 and 2147483647'. The 'Email' field also has a red border and the error message 'may not be empty'. Below the fields is a 'Birth Date' field with the error message 'may not be null'. At the bottom is a blue 'SUBMIT' button with a white arrow.

The next thing we need to do is to customize the error messages to reflect the business rules of our application in a better way.

Remember that Spring Boot takes care of creating a message source bean for us? The default location for this message source is in `src/main/resources/messages.properties`.

Let's create such a bundle, and add the following text:

```

Size.profileForm.twitterHandle=Please type in your twitter user name
Email.profileForm.email=Please specify a valid email address
NotEmpty.profileForm.email=Please specify your email address
PastLocalDate.profileForm.birthDate=Please specify a real birth date
NotNull.profileForm.birthDate=Please specify your birth date

typeMismatch.birthDate = Invalid birth date format.

```



It can be very handy in development to configure the message source to always reload our bundles. Add the following property to `application.properties`:

```
spring.messages.cache-seconds=0
```

0 means always reload, whereas -1 means never reload.

The class responsible for resolving the error messages in Spring is `DefaultMessageCodesResolver`. In the case of field validation, this class tries to resolve the following messages in the given order:

- `code + "." + object name + "." + field`
- `code + "." + field`
- `code + "." + field type`
- `code`

In the preceding rules, the code part can be two things: an annotation type such as `Size` or `Email`, or an exception code such as `typeMismatch`. Remember when we got an exception caused by an incorrect date format? The associated error code was indeed `typeMismatch`.

With the preceding messages, we chose to be very specific. A good practice is to define default messages as follows:

```
Size={0} field must be between {2} and {1} characters long
typeMismatch.java.util.Date = Invalid date format.
```

Note the placeholders; each validation error has a number of arguments associated with it.

The last way to declare error messages would involve defining the error message directly in the validation annotations as follows:

```
@Size(min = 2, message = "Please specify a valid twitter handle")
private String twitterHandle;
```

However, the downside of this method is that it is not compatible with internationalization.

Custom annotation for validation

For Java dates, there is an annotation called `@Past`, which ensures that a date is from the past.

We don't want our user to pretend they are coming from the future, so we need to validate the birth date. To do this, we will define our own annotation in the `date` package:

```
package masterSpringMvc.date;

import javax.validation.Constraint;
import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;
import javax.validation.Payload;
import java.lang.annotation.*;
import java.time.LocalDate;

@Target({ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = PastLocalDate.PastValidator.class)
@Documented
public @interface PastLocalDate {
    String message() default "{javax.validation.constraints.Past.message}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    class PastValidator implements ConstraintValidator<PastLocalDate,
    LocalDate> {
        public void initialize(PastLocalDate past) {
        }

        public boolean isValid(LocalDate localDate,
        ConstraintValidatorContext context) {
            return localDate == null || localDate.isBefore(LocalDate.
        now());
        }
    }
}
```

Simple isn't it? This code will verify that our date is really from the past.

We can now add it to the `birthDate` field in the profile form:

```
@NotNull  
@PastLocalDate  
private LocalDate birthDate;
```

Internationalization

Internationalization, frequently abbreviated i18n, is the process of designing an application that can be translated into various languages.

This generally involves placing translations in properties bundles with their names suffixed with the target locale, for instance, the `messages_en.properties`, `messages_en_US.properties`, and `messages_fr.properties` files.

The correct property bundle is resolved by trying the most specific locale first and then falling back to the less specific ones.

For U.S English, if you try to get a translation from a bundle named `x`, the application would first look in the `x_en_US.properties` file, then the `x_en.properties` file, and finally, the `x.properties` file.

The first thing we will do is translate our error messages into French. To do this, we will rename our existing `messages.properties` file to `messages_en.properties`.

We will also create a second bundle named `messages_fr.properties`:

```
Size.profileForm.twitterHandle=Veuillez entrer votre identifiant  
Twitter  
Email.profileForm.email=Veuillez spécifier une adresse mail valide  
NotEmpty.profileForm.email=Veuillez spécifier votre adresse mail  
PastLocalDate.profileForm.birthDate=Veuillez donner votre vraie date  
de naissance  
NotNull.profileForm.birthDate=Veuillez spécifier votre date de  
naissance  
  
typeMismatch.birthDate = Date de naissance invalide.
```

We saw in *Chapter 1, Setting Up a Spring Web Application in No Time* that by default, Spring Boot uses a fixed `LocaleResolver` interface. The `LocaleResolver` is a simple interface with two methods:

```
public interface LocaleResolver {  
  
    Locale resolveLocale(HttpServletRequest request);
```

```
void setLocale(HttpServletRequest request, HttpServletResponse
response, Locale locale);
}
```

Spring provides a bunch of implementations of this interface, such as `FixedLocaleResolver`. This local resolver is very simple; we can configure the application locale via a property and cannot change it once it is defined. To configure the locale of our application, let's add the following property to our application `properties` file:

```
spring.mvc.locale=fr
```

This will add our validation messages in French.

If we take a look at the different `LocaleResolver` interfaces that are bundled in Spring MVC, we will see the following:

- `FixedLocaleResolver`: This fixes the locale defined in configuration. It cannot be changed once fixed.
- `CookieLocaleResolver`: This allows the locale to be retrieved and saved in a cookie.
- `AcceptHeaderLocaleResolver`: This uses the HTTP header sent by the user's browser to find the locale.
- `SessionLocaleResolver`: This finds and stores the locale in an HTTP session.

These implementations cover a number of use cases, but in a more complex application one might implement `LocaleResolver` directly to allow more complex logic such as fetching the locale from the database and falling back to browser locale, for instance.

Changing the locale

In our application, the locale is linked to the user. We will save their profile in session.

We will allow the user to change the language of the site using a small menu. That's why we will use the `SessionLocaleResolver`. Let's edit `webConfiguration` once more:

```
package masterSpringMvc.config;

import masterSpringMvc.date.USLocalDateFormatter;
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.
WebMvcConfigurerAdapter;
import org.springframework.web.servlet.i18n.LocaleChangeInterceptor;
import org.springframework.web.servlet.i18n.SessionLocaleResolver;

import java.time.LocalDate;

@Configuration
public class WebConfiguration extends WebMvcConfigurerAdapter {

    @Override
    public void addFormatters(FormatterRegistry registry) {
        registry.addFormatterForFieldType(LocalDate.class, new
USLocalDateFormatter());
    }

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

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

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

We declared a `LocaleChangeInterceptor` bean as a Spring MVC interceptor. It will intercept any request made to Controller and check for the `lang` query parameter. For instance, navigating to `http://localhost:8080/profile?lang=fr` would cause the locale to change.



Spring MVC Interceptors can be compared to Servlet filters in a web application. Interceptors allow custom preprocessing, skipping the execution of a handler, and custom post-processing. Filters are more powerful, for example, they allow for exchanging the request and response objects that are handed down the chain. Filters are configured in a web.xml file, while interceptors are declared as beans in the application context.

Now, we can change the locale by entering the correct URL ourselves, but it would be better to add a navigation bar allowing the user to change the language. We will modify the default layout (templates/layout/default.html) to add a drop-down menu:

```

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout">
<head>
    <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8"/>
    <meta name="viewport" content="width=device-width, initial-
scale=1, maximum-scale=1.0, user-scalable=no"/>
    <title>Default title</title>

    <link href="/webjars/materializecss/0.96.0/css/materialize.css"
          type="text/css" rel="stylesheet" media="screen,projection"/>
</head>
<body>

    <ul id="lang-dropdown" class="dropdown-content">
        <li><a href="?lang=en_US">English</a></li>
        <li><a href="?lang=fr">French</a></li>
    </ul>
    <nav>
        <div class="nav-wrapper indigo">
            <ul class="right">
                <li><a class="dropdown-button" href="#" data-
activates="lang-dropdown"><i class="mdi-action-language right"></i>
Lang</a></li>
            </ul>
        </div>
    </nav>

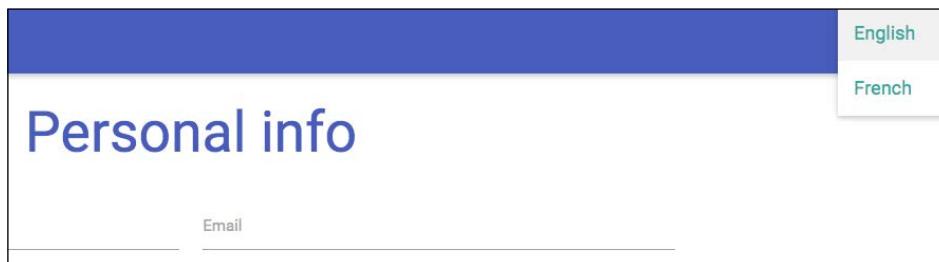
    <section layout:fragment="content">
        <p>Page content goes here</p>

```

```
</section>

<script src="/webjars/jquery/2.1.4/jquery.js"></script>
<script src="/webjars/materializecss/0.96.0/js/materialize.js"></script>
<script type="text/javascript">
    $(".dropdown-button").dropdown();
</script>
</body>
</html>
```

This will allow the user to choose between the two supported languages.



Translating the application text

The last thing we need to do in order to have a fully bilingual application is to translate the titles and labels of our application. To do this, we will edit our web pages and use the `th:text` attribute, for instance, in `profilePage.html`:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      layout:decorator="layout/default">
<head lang="en">
    <title>Your profile</title>
</head>
<body>
<div class="row" layout:fragment="content">

    <h2 class="indigo-text center" th:text="#{profile.title}">Personal
    info</h2>

    <form th:action="@{/profile}" th:object="${profileForm}"
          method="post" class="col m8 s12 offset-m2">
```

```

<div class="row">
    <div class="input-field col s6">
        <input th:field="${profileForm.twitterHandle}"
id="twitterHandle" type="text" th:errorclass="invalid"/>
        <label for="twitterHandle" th:text="#{twitter.
handle}">Twitter handle</label>

        <div th:errors="*{twitterHandle}" class="red-
text">Error</div>
    </div>
    <div class="input-field col s6">
        <input th:field="${profileForm.email}" id="email"
type="text" th:errorclass="invalid"/>
        <label for="email" th:text="#{email}">Email</label>

        <div th:errors="*{email}" class="red-text">Error</div>
    </div>
</div>
<div class="row">
    <div class="input-field col s6">
        <input th:field="${profileForm.birthDate}"
id="birthDate" type="text" th:errorclass="invalid"/>
        <label for="birthDate" th:text="#{birthdate}" th:place
holder="${dateFormat}">Birth Date</label>

        <div th:errors="*{birthDate}" class="red-text">Error</
div>
    </div>
    <div class="row s12 center">
        <button class="btn indigo waves-effect waves-light"
type="submit" name="save" th:text="#{submit}">Submit
        <i class="mdi-content-send right"></i>
    </button>
    </div>
</form>
</div>
</body>
</html>

```

The `th:text` attribute will replace the contents of a HTML element with an expression. Here, we use the `#{}` syntax, which indicates we want to display a message coming from a property source like `messages.properties`.

Let's add the corresponding translations to our English bundle:

```
NotEmpty.profileForm.tastes=Please enter at least one thing
profile.title=Your profile
twitter.handle=Twitter handle
email=Email
birthdate=Birth Date
tastes.legend=What do you like?
remove=Remove
taste.placeholder=Enter a keyword
add.taste=Add taste
submit=Submit
```

Now to the French ones:

```
NotEmpty.profileForm.tastes=Veuillez saisir au moins une chose
profile.title=Votre profil
twitter.handle=Pseudo twitter
email=Email
birthdate=Date de naissance
tastes.legend=Quels sont vos goûts ?
remove=Supprimer
taste.placeholder=Entrez un mot-clé
add.taste=Ajouter un centre d'intérêt
submit=Envoyer
```

Some of the translations are not used yet, but will be used in just a moment. Et voilà! The French market is ready for the Twitter search flood.

A list in a form

We now want the user to enter a list of "tastes", which are, in fact, a list of keywords we will use to search tweets.

A button will be displayed, allowing our user to enter a new keyword and add it to a list. Each item of this list will be an editable input text and will be removable thanks to a remove button:

The form is a horizontal layout. At the top is a text input for 'Birth Date'. Below it is a legend box with the text 'What do you like?'. Under the legend is a teal 'ADD TASTE' button. The main content area contains two entries: 'spring' and 'java', each with a 'REMOVE' button to its right. Below these entries is a 'SUBMIT' button.

Handling list data in a form can be a chore with some frameworks. However, with Spring MVC and Thymeleaf it is relatively straightforward, when you understand the principle.

Add the following lines in the `profilePage.html` file right below the row containing the birth date, and just over the submit button:

```

<fieldset class="row">
    <legend th:text="#{tastes.legend}">What do you like?</legend>
    <button class="btn teal" type="submit" name="addTaste"
    th:text="#{add.taste}">Add taste
        <i class="mdi-content-add left"></i>
    </button>

    <div th:errors="*{tastes}" class="red-text">Error</div>

    <div class="row" th:each="row, rowStat : *{tastes}">
        <div class="col s6">
            <input type="text" th:field="*{tastes[__${rowStat.
            index}__]}>" th:placeholder="#{taste.placeholder}">
        </div>

        <div class="col s6">
            <button class="btn red" type="submit" name="removeTaste"
            th:value="${rowStat.index}" th:text="#{remove}">Remove
                <i class="mdi-action-delete right waves-effect"></i>
            </button>
        </div>
    </div>
</fieldset>

```

The purpose of this snippet is to iterate over the `tastes` variable of our `LoginForm`. This can be achieved with the `th:each` attribute, which looks a lot like a `for...in` loop in java.

Compared to the search result loop we saw earlier, the iteration is stored in two variables instead of one. The first one will actually contain each row of the data. The `rowStat` variable will contain additional information on the current state of the iteration.

The strangest thing in the new piece of code is:

```
th:field="*{tastes[__${rowStat.index}__]}"
```

This is quite a complicated syntax. You could come up with something simpler on your own, such as:

```
th:field="*{tastes[rowStat.index]}"
```

Well, that wouldn't work. The `${rowStat.index}` variable, which represents the current index of the iteration loop, needs to be evaluated before the rest of the expression. To achieve this, we need to use preprocessing.

The expression surrounded by double underscores will be preprocessed, which means that it will be processed before the normal processing phase, allowing it to be evaluated twice.

There are two new submit buttons on our form now. They all have a name. The global submit button we had earlier is called `save`. The two new buttons are called `addTaste` and `removeTaste`.

On the controller side, this will allow us to easily discriminate the different actions coming from our form. Let's add two new actions to our `ProfileController`:

```
@Controller
public class ProfileController {

    @ModelAttribute("dateFormat")
    public String localeFormat(Locale locale) {
        return USLocalDateFormatter.getPattern(locale);
    }

    @RequestMapping("/profile")
    public String displayProfile(ProfileForm profileForm) {
        return "profile/profilePage";
    }
}
```

```

    @RequestMapping(value = "/profile", params = {"save"}, method =
RequestMethod.POST)
    public String saveProfile(@Valid ProfileForm profileForm,
BindingResult bindingResult) {
        if (bindingResult.hasErrors()) {
            return "profile/profilePage";
        }
        System.out.println("save ok" + profileForm);
        return "redirect:/profile";
    }

    @RequestMapping(value = "/profile", params = {"addTaste"})
    public String addRow(ProfileForm profileForm) {
        profileForm.getTastes().add(null);
        return "profile/profilePage";
    }

    @RequestMapping(value = "/profile", params = {"removeTaste"})
    public String removeRow(ProfileForm profileForm,
HttpServletRequest req) {
        Integer rowId = Integer.valueOf(req.
getParameter("removeTaste"));
        profileForm.getTastes().remove(rowId.intValue());
        return "profile/profilePage";
    }
}

```

We added a `param` parameter to each of our post actions to differentiate them. The one we had previously is now bound to the `save` parameter.

When we click on a button, its name will automatically be added to the form data sent by the browser. Note that we specified a particular value with the remove button: `th:value="${rowStat.index}"`. This attribute will indicate which value the associated parameter should specifically take. A blank value will be sent if this attribute is not present. This means that when we click on the remove button, a `removeTaste` parameter will be added to the `POST` request, containing the index of the row we would like to remove. We can then get it back into the Controller with the following code:

```
Integer rowId = Integer.valueOf(req.getParameter("removeTaste"));
```

The only downside with this method is that the whole form data will be sent every time we click on the button, even if it is not strictly required. Our form is small enough, so a tradeoff is acceptable.

That's it! The form is now complete, with the possibility of adding one or more tastes.

Client validation

As a little bonus, client-side validation has become very easy nowadays with the HTML5 form validation specification. If your target browsers are Internet Explorer 10 and above, adding client-side validation is as easy as specifying the correct input type instead of just using text.

By adding the client-side validation, we can prevalidate the form and avoid overloading the server with requests that we know are incorrect. More information on the client-side validation specification is available at <http://caniuse.com/#search=validation>.

We can modify our inputs to enable simple client-side validation. The previous inputs, as shown in the following code:

```
<input th:field="${profileForm.twitterHandle}" id="twitterHandle"
type="text" th:errorclass="invalid"/>
<input th:field="${profileForm.email}" id="email" type="text"
th:errorclass="invalid"/>
<input th:field="${profileForm.birthDate}" id="birthDate" type="text"
th:errorclass="invalid"/>
<input type="text" th:field="*{tastes[__${rowStat.index}__]} "
th:placeholder="#{taste.placeholder}"/>
```

This becomes:

```
<input th:field="${profileForm.twitterHandle}" id="twitterHandle"
type="text" required="required" th:errorclass="invalid"/>
<input th:field="${profileForm.email}" id="email" type="email"
required="required" th:errorclass="invalid"/>
<input th:field="${profileForm.birthDate}" id="birthDate" type="text"
required="required" th:errorclass="invalid"/>
<input type="text" required="required" th:field="*{tastes[__${rowStat.
index}__]} " th:placeholder="#{taste.placeholder}"/>
```

With this method, your browser will detect when the form is submitted and validate each attribute according to its type. The **required** attribute forces the user to enter a nonblank value. The **email** type enforces basic e-mail validation rules on the corresponding field.

The form shows two fields: 'Twitter handle' with value 'geowarin' and 'Email' with value 'geo'. A validation message box is displayed, stating: 'Please include an '@' in the email address. 'geo' is missing an '@'.'

Other types of validators also exist. Take a look at <http://www.the-art-of-web.com/html/html5-form-validation>.

The downside of this method is that our add taste and remove taste buttons will now trigger validation. To fix this, we need to include a script at the bottom of the default layout, right after the jQuery declaration.

However, it would be best to include it only on the profile page. To do this, we can add a new fragment section in the `layout/default.html` page, just before the end of the body tag:

```
<script type="text/javascript" layout:fragment="script">
</script>
```

This will allow us to include an additional script on each page if needed.

Now, we can add the following script to our profile page, just before closing the body tag:

```
<script layout:fragment="script">
  $('button').bind('click', function(e) {
    if (e.currentTarget.name === 'save') {
      $(e.currentTarget.form).removeAttr('novalidate');
    } else {
      $(e.currentTarget.form).attr('novalidate', 'novalidate');
    }
  });
</script>
```

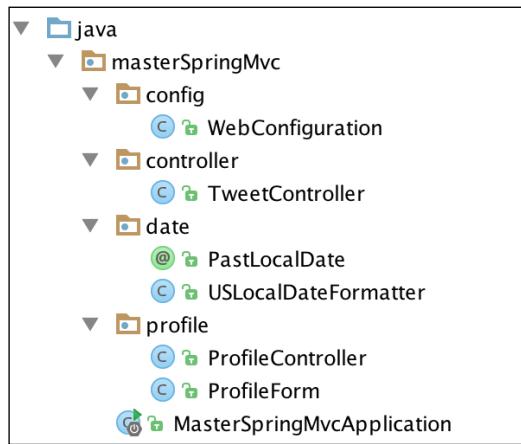
Form validation won't be triggered when a `novalidate` attribute is present on the form. This little script will dynamically remove the `novalidate` attribute if the action of the form is named `save` if the name of the input is different, the `novalidate` attribute will always be added. Validation will thus be triggered only by the `save` button.

The check point

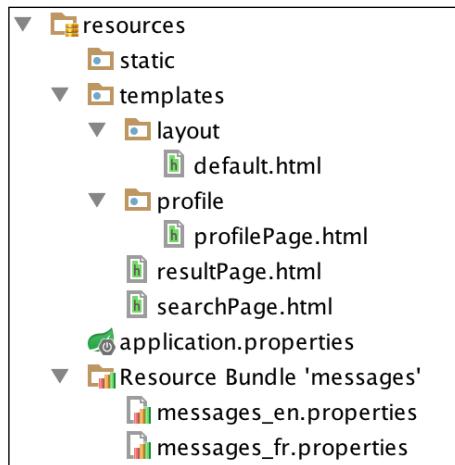
Before moving on to the next chapter, let's check whether everything is in the right place.

In the Java sources, you should have the following:

- A new controller, the `ProfileController`
- Two new classes related to date: a date formatter and an annotation to validate `LocalDates`
- A new `WebConfiguration` folder to customize Spring MVC's configuration



In the resources, you should have a new template inside the profile directory and two new bundles:



Summary

In this chapter, you learned how to make a complete form. We created a model using Java 8 dates, and you learned how to format information coming from the user and display it accordingly.

We ensured that the form was filled with valid information, with validator annotations, including our own. Also, we prevented obviously incorrect information from even hitting the server by including some client-side validation very easily.

Finally, we even translated the whole application into English and French, date formats included!

In the next chapter, we will build a space where the users will be able to upload their pictures and learn more about error handling in Spring MVC applications.

4

File Upload and Error Handling

In this chapter, we will enable our user to upload a profile picture. We will also see how to handle errors in Spring MVC.

Uploading a file

We will now make it possible for our user to upload a profile picture. This will be available from the profile page later on, but for now, we will simplify things and create a new page in the templates directory under `profile/uploadPage.html`:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      layout:decorator="layout/default">
<head lang="en">
    <title>Profile Picture Upload</title>
</head>
<body>
<div class="row" layout:fragment="content">

    <h2 class="indigo-text center">Upload</h2>

    <form th:action="@{/upload}" method="post" enctype="multipart/form-data" class="col m8 s12 offset-m2">

        <div class="input-field col s6">
            <input type="file" id="file" name="file"/>
        </div>
```

```
<div class="col s6 center">
    <button class="btn indigo waves-effect waves-light"
type="submit" name="save" th:text="#{submit}">Submit
        <i class="mdi-content-send right"></i>
    </button>
</div>
</form>
</div>
</body>
</html>
```

Not much to see besides the `enctype` attribute on the form. The file will be sent by the `POST` method to the upload URL. We will now create the corresponding controller right beside `ProfileController` in the `profile` package:

```
package masterSpringMvc.profile;

import org.apache.tomcat.util.http.fileupload.IOUtils;
import org.springframework.core.io.FileSystemResource;
import org.springframework.core.io.Resource;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.multipart.MultipartFile;

import java.io.*;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

@Controller
public class PictureUploadController {
    public static final Resource PICTURES_DIR = new
    FileSystemResource("./pictures");

    @RequestMapping("upload")
    public String uploadPage() {
        return "profile/uploadPage";
    }

    @RequestMapping(value = "/upload", method = RequestMethod.POST)
    public String onUpload(MultipartFile file) throws IOException {
        String filename = file.getOriginalFilename();
```

```

        File tempFile = File.createTempFile("pic",
        getFileExtension(filename), PICTURES_DIR.getFile());

        try (InputStream in = file.getInputStream();
            OutputStream out = new FileOutputStream(tempFile)) {
            IOUtils.copy(in, out);
        }

        return "profile/uploadPage";
    }

    private static String getFileExtension(String name) {
        return name.substring(name.lastIndexOf("."));
    }
}

```

The first thing this code will do is create a temporary file in the pictures directory, which can be found inside the project's root folder; so, ensure that it exists. In Java, a temporary file is just a commodity to obtain a unique file identifier on the filesystem. It is up to the user to optionally delete it.

Create a pictures directory at the root of the project and add an empty file called `.gitkeep` to ensure that you can commit it in Git.


Empty directories in Git

Git is file-based and it is not possible to commit an empty directory. A common workaround is to commit an empty file, such as `.gitkeep`, in a directory to force Git to keep it under version control.

The file uploaded by the user will be injected as a `MultipartFile` interface in our controller. This interface provides several methods to get the name of the file, its size, and its contents.

The method that particularly interests us here is `getInputStream()`. We will indeed copy this stream to a `fileOutputStream` method, thanks to the `IOUtils.copy` method. The code to write an input stream to an output stream is pretty boring, so it's handy to have the Apache Utils in the classpath (it is part of the `tomcat-embedded-core.jar` file).

We make heavy use of the pretty cool Spring and Java 7 NIO features:

- The `Resource` class of `String` is a utility class that represents an abstraction of resources that can be found in different ways
- The `try...with` block will automatically close our streams even in the case of an exception, removing the boilerplate of writing a `finally` block

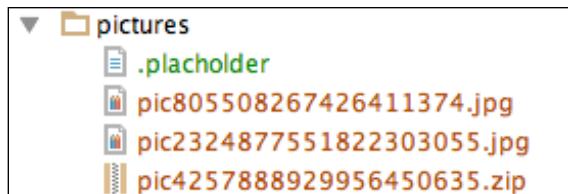
With the preceding code, any file uploaded by the user will be copied into the `pictures` directory.

There are a handful of properties available in Spring Boot to customize file upload. Take a look at the `MultipartProperties` class.

The most interesting ones are:

- `multipart.maxFileSize`: This defines the maximum file size allowed for the uploaded files. Trying to upload a bigger one will result in a `MultipartException` class. The default value is `1Mb`.
- `multipart.maxRequestSize`: This defines the maximum size of the multipart request. The default value is `10Mb`.

The defaults are good enough for our application. After a few uploads, our picture directory will look like this:



Wait! Somebody uploaded a ZIP file! I cannot believe it. We better add some checks in our controller to ensure that the uploaded files are real images:

```
package masterSpringMvc.profile;

import org.apache.tomcat.util.http.fileupload.IOUtils;
import org.springframework.core.io.FileSystemResource;
import org.springframework.core.io.Resource;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.multipart.MultipartFile;
import org.springframework.web.servlet.mvc.support.RedirectAttributes;
```

```
import java.io.*;

@Controller
public class PictureUploadController {
    public static final Resource PICTURES_DIR = new
FileSystemResource("./pictures");

    @RequestMapping("upload")
    public String uploadPage() {
        return "profile/uploadPage";
    }

    @RequestMapping(value = "/upload", method = RequestMethod.POST)
    public String onUpload(MultipartFile file, RedirectAttributes
redirectAttrs) throws IOException {

        if (file.isEmpty() || !isImage(file)) {
            redirectAttrs.addFlashAttribute("error", "Incorrect file.
Please upload a picture.");
            return "redirect:/upload";
        }

        copyFileToPictures(file);

        return "profile/uploadPage";
    }

    private Resource copyFileToPictures(MultipartFile file) throws
IOException {
        String fileExtension = getFileExtension(file.
getOriginalFilename());
        File tempFile = File.createTempFile("pic", fileExtension,
PICTURES_DIR.getFile());
        try (InputStream in = file.getInputStream();
            OutputStream out = new FileOutputStream(tempFile)) {

            IOUtils.copy(in, out);
        }
        return new FileSystemResource(tempFile);
    }

    private boolean isImage(MultipartFile file) {
        return file.getContentType().startsWith("image");
    }
}
```

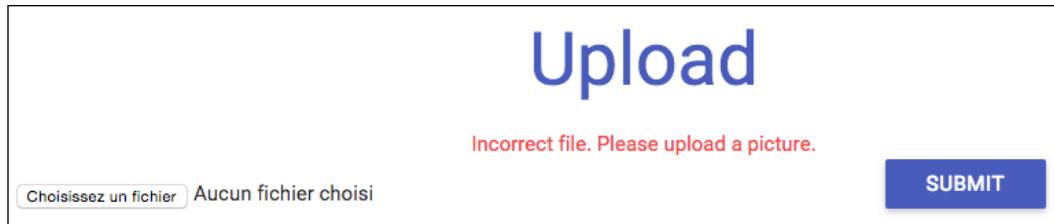
```
private static String getFileExtension(String name) {  
    return name.substring(name.lastIndexOf("."));  
}  
}
```

Pretty easy! The `getContentType()` method returns the **Multipurpose Internet Mail Extensions (MIME)** type of the file. It will be `image/png`, `image/jpg`, and so on. So we just have to check if the MIME type starts with "image".

We added an error message to the form so we should add something in our web page to display it. Place the following code just under the title in the `uploadPage`:

```
<div class="col s12 center red-text" th:text="${error}"  
th:if="${error}">  
    Error during upload  
</div>
```

The next time you try to upload a ZIP file, you will get an error! This is shown in the following screenshot:



Writing an image to the response

The uploaded images are not served from the static directories. We will need to take special measures to display them in our web page.

Let's add the following lines to our upload page, just above the form:

```
<div class="col m8 s12 offset-m2">  
      
</div>
```

This will try and get the image from our controller. Let's add the corresponding method to the `PictureUploadController` class:

```
@RequestMapping(value = "/uploadedPicture")  
public void getUploadedPicture(HttpServletRequest response) throws  
IOException {
```

```

ClassPathResource classPathResource = new ClassPathResource("/images/anonymous.png");
response.setHeader("Content-Type", URLConnection.guessContentTypeFromName(classPathResource.getFilename()));
IOUtils.copy(classPathResource.getInputStream(), response.getOutputStream());
}

```

This code will write an image found in the `src/main/resources/images/anonymous.png` directory directly to the response! How exciting!

If we go to our page again, we will see the following image:



[ I found the anonymous user image on iconmonstr (<http://iconmonstr.com/user-icon>) and downloaded it as a 128 x 128 PNG file.]

Managing upload properties

A good thing to do at this point is to allow the configuration of the upload directory and the path to the anonymous user image through the `application.properties` file.

Let's create a `PicturesUploadProperties` class inside a newly created `config` package:

```

package masterSpringMvc.config;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.core.io.DefaultResourceLoader;
import org.springframework.core.io.Resource;

import java.io.IOException;

```

File Upload and Error Handling

```
@ConfigurationProperties(prefix = "upload.pictures")
public class PictureUploadProperties {
    private Resource uploadPath;
    private Resource anonymousPicture;

    public Resource getAnonymousPicture() {
        return anonymousPicture;
    }

    public void setAnonymousPicture(String anonymousPicture) {
        this.anonymousPicture = new DefaultResourceLoader().
        getResource(anonymousPicture);
    }

    public Resource getUploadPath() {
        return uploadPath;
    }

    public void setUploadPath(String uploadPath) {
        this.uploadPath = new DefaultResourceLoader().
        getResource(uploadPath);
    }
}
```

In this class, we make use of the Spring Boot ConfigurationProperties. This will tell Spring Boot to automatically map properties found in the classpath (by default, in the application.properties file) in a type-safe fashion.

Notice that we defined setters taking 'String's as arguments but are at liberty to let the getters return any type is the most useful.

We now need to add the PictureUploadProperties class to our configuration:

```
@SpringBootApplication
@EnableConfigurationProperties({PictureUploadProperties.class})
public class MasterSpringMvc4Application extends
WebMvcConfigurerAdapter {
    // code omitted
}
```

We can now add the properties' values inside the application.properties file:

```
upload.pictures.uploadPath=file:./pictures
upload.pictures.anonymousPicture=classpath:/images/anonymous.png
```

Because we use Spring's `DefaultResourceLoader` class, we can use prefixes such as `file:` or `classpath:` to specify where our resources can be found.

This would be the equivalent of creating a `FileSystemResource` class or a `ClassPathResource` class.

This approach also has the advantage of documenting the code. We can easily see that the picture directory will be found in the application root, whereas the anonymous picture will be found in the classpath.

That's it. We can now use our properties inside our controller. The following are the relevant parts of the `PictureUploadController` class:

```
package masterSpringMvc.profile;

import masterSpringMvc.config.PictureUploadProperties;
import org.apache.tomcat.util.http.fileupload.IOUtils;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.io.Resource;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.multipart.MultipartFile;
import org.springframework.web.servlet.mvc.support.RedirectAttributes;

import javax.servlet.http.HttpServletResponse;
import java.io.*;
import java.net.URLConnection;

@Controller
public class PictureUploadController {
    private final Resource picturesDir;
    private final Resource anonymousPicture;

    @Autowired
    public PictureUploadController(PictureUploadProperties
        uploadProperties) {
        picturesDir = uploadProperties.getUploadPath();
        anonymousPicture = uploadProperties.getAnonymousPicture();
    }

    @RequestMapping(value = "/uploadedPicture")
    public void getUploadedPicture(HttpServletResponse response)
        throws IOException {
```

```
    response.setHeader("Content-Type", URLConnection.guessContentTypeFromName(anonymousPicture.getFilename()));
    IOUtils.copy(anonymousPicture.getInputStream(), response.getOutputStream());
}

private Resource copyFileToPictures(MultipartFile file) throws IOException {
    String fileExtension = getFileExtension(file.getOriginalFilename());
    File tempFile = File.createTempFile("pic", fileExtension,
picturesDir.getFile());
    try (InputStream in = file.getInputStream();
        OutputStream out = new FileOutputStream(tempFile)) {

        IOUtils.copy(in, out);
    }
    return new FileSystemResource(tempFile);
}
// The rest of the code remains the same
}
```

At this point, if you launch your application again, you will see that the result hasn't changed. The anonymous picture is still displayed and the pictures uploaded by our users still end up in the pictures directory at the project root.

Displaying the uploaded picture

It would be nice to display the user's picture now, wouldn't it? To do this, we will add a model attribute to our PictureUploadController class:

```
@ModelAttribute("picturePath")
public Resource picturePath() {
    return anonymousPicture;
}
```

We can now inject it to retrieve its value when we serve the uploaded picture:

```
@RequestMapping(value = "/uploadedPicture")
public void getUploadedPicture(HttpServletRequest response, @
ModelAttribute("picturePath") Path picturePath) throws IOException {
    response.setHeader("Content-Type", URLConnection.guessContentTypeFromName(picturePath.toString()));
    Files.copy(picturePath, response.getOutputStream());
}
```

The `@ModelAttribute` annotation is a handy way to create model attributes with an annotated method. They can then be injected with the same annotation into controller methods. With this code, a `picturePath` parameter will be available in the model as long as we are not redirected to another page. Its default value is the anonymous picture we defined in our properties.

We need to update this value when the file is uploaded. Update the `onUpload` method:

```
@RequestMapping(value = "/upload", method = RequestMethod.POST)
public String onUpload(MultipartFile file, RedirectAttributes
redirectAttrs, Model model) throws IOException {

    if (file.isEmpty() || !isImage(file)) {
        redirectAttrs.addFlashAttribute("error", "Incorrect file.
Please upload a picture.");
        return "redirect:/upload";
    }

    Resource picturePath = copyFileToPictures(file);
    model.addAttribute("picturePath", picturePath);

    return "profile/uploadPage";
}
```

By injecting the model, we can update the `picturePath` parameter after the upload is complete.

Now, the problem is that our two methods, `onUpload` and `getUploadedPicture`, will occur in different requests. Unfortunately, the model attributes will be reset between each.

That's why we will define the `picturePath` parameter as a session attribute. We can do this by adding another annotation to our controller class:

```
@Controller
@SessionAttributes("picturePath")
public class PictureUploadController {
```

Phew! That's a lot of annotations just to handle a simple session attribute. You will get the following output:



This approach makes code composition really easy. Plus, we didn't use `HttpServletRequest` or `HttpSession` directly. Moreover, our object can be typed easily.

Handling file upload errors

It must have certainly occurred to my attentive readers that our code is susceptible to throw two kinds of exceptions:

- `IOException`: This error is thrown if something bad happens while writing the file to disk.
- `MultipartException`: This error is thrown if an error occurs while uploading the file. For instance, when the maximum file size is exceeded.

This will give us a good opportunity to look at two ways of handling exceptions in Spring:

- Using the `@ExceptionHandler` annotation locally in a controller method
- Using a global exception handler defined at the Servlet container level

Let's handle `IOException` with the `@ExceptionHandler` annotation inside our `PictureUploadController` class by adding the following method:

```
@ExceptionHandler(IOException.class)
public ModelAndView handleIOException(IOException exception) {
    ModelAndView modelAndView = new ModelAndView("profile/
uploadPage");
```

```

        modelAndView.addObject("error", exception.getMessage());
        return modelAndView;
    }

```

This is a simple yet powerful approach. This method will be called every time an `IOException` is thrown in our controller.

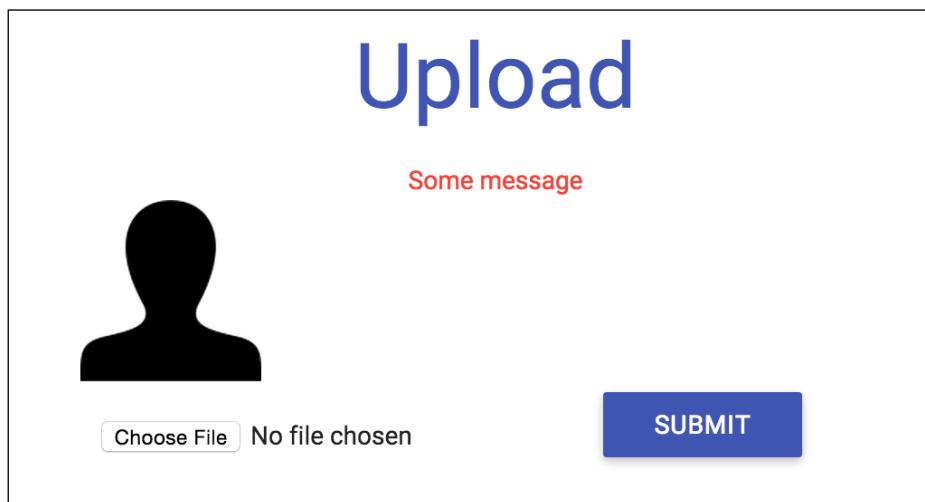
In order to test the exception handler, since making the Java IO code throw an exception can be tricky, just replace the `onUpload` method body during the test:

```

@RequestMapping(value = "/upload", method = RequestMethod.POST)
public String onUpload(MultipartFile file, RedirectAttributes
redirectAttrs, Model model) throws IOException {
    throw new IOException("Some message");
}

```

After this change, if we try to upload a picture, we will see the error message of this exception displayed on the upload page:



Now, we will handle the `MultipartException`. This needs to happen at the Servlet container level (that is, at the Tomcat level), as this exception is not thrown directly by our controller.

We will need to add a new `EmbeddedServletContainerCustomizer` bean to our configuration. Add this method to the `WebConfiguration` class:

```

@Bean
public EmbeddedServletContainerCustomizer containerCustomizer() {
    EmbeddedServletContainerCustomizer

```

```
embeddedServletContainerCustomizer = new
EmbeddedServletContainerCustomizer() {
    @Override
    public void customize(ConfigurableEmbeddedServletContainer
container) {
        container.addErrorPages(new ErrorPage(MultipartException.
class, "/uploadError"));
    }
};
return embeddedServletContainerCustomizer;
}
```

This is a little verbose. Note that `EmbeddedServletContainerCustomizer` is an interface that contains a single method; it can therefore be replaced by a lambda expression:

```
@Bean
public EmbeddedServletContainerCustomizer containerCustomizer() {
    EmbeddedServletContainerCustomizer
embeddedServletContainerCustomizer
        = container -> container.addErrorPages(new
ErrorPage(MultipartException.class, "/uploadError"));
    return embeddedServletContainerCustomizer;
}
```

So, let's just write the following:

```
@Bean
public EmbeddedServletContainerCustomizer containerCustomizer() {
    return container -> container.addErrorPages(new
ErrorPage(MultipartException.class, "/uploadError"));
}
```

This code creates a new error page, which will be called when a `MultipartException` happens. It can also be mapped to an HTTP status. The `EmbeddedServletContainerCustomizer` interface has many other features that will allow the customization of the Servlet container in which our application runs. Visit <http://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-developing-web-applications.html#boot-features-customizing-embedded-containers> for more information.

We now need to handle this `uploadError` URL in our `PictureUploadController` class:

```
@RequestMapping("uploadError")
public ModelAndView onUploadError(HttpServletRequest request) {
    ModelAndView modelAndView = new ModelAndView("uploadPage");
```

```

        modelAndView.addObject("error", request.getAttribute(WebUtils.
        ERROR_MESSAGE_ATTRIBUTE));
        return modelAndView;
    }
}

```

The error pages defined in a Servlet environment contain a number of interesting attributes that will help debug the error:

Attribute	Description
javax.servlet.error.status_code	This is the HTTP status code of the error.
javax.servlet.error.exception_type	This is the exception class.
javax.servlet.error.message	This is the message of the exception thrown.
javax.servlet.error.request_uri	This is the URI on which the exception occurred.
javax.servlet.error.exception	This is the actual exception.
javax.servlet.error.servlet_name	This is the name of the Servlet that caught the exception.

All these attributes are conveniently accessible on the `WebUtils` class of Spring Web.

If someone tries to upload too big a file, they will get a very clear error message.

You can now test that the error is handled correctly by uploading a really big file (> 1Mb) or setting the `multipart.maxFileSize` property to a lower value: 1kb for instance

Upload



Request processing failed; nested exception is org.springframework.web.multipart.MultipartException:
 Could not parse multipart servlet request; nested exception is java.lang.IllegalStateException:
 org.apache.tomcat.util.http.fileupload.FileUploadBase\$FileSizeLimitExceededException: The field file
 exceeds its maximum permitted size of 1048576 bytes.

Aucun fichier choisi

Translating the error messages

It is really good for a developer to see the exceptions thrown by the application. However, for our users, they bear little value. We will therefore translate them. In order to do that, we have to inject our application's `MessageSource` class into our controller's constructor:

```
private final MessageSource messageSource;

@Autowired
public PictureUploadController(PictureUploadProperties
uploadProperties, MessageSource messageSource) {
    picturesDir = uploadProperties.getUploadPath();
    anonymousPicture = uploadProperties.getAnonymousPicture();
    this.messageSource = messageSource;
}
```

Now, we can retrieve messages from our messages bundle:

```
@ExceptionHandler(IOException.class)
public ModelAndView handleIOException(Locale locale) {
    ModelAndView modelAndView = new ModelAndView("profile/
uploadPage");
    modelAndView.addObject("error", messageSource.getMessage("upload.
io.exception", null, locale));
    return modelAndView;
}

@RequestMapping("uploadError")
public ModelAndView onUploadError(Locale locale) {
    ModelAndView modelAndView = new ModelAndView("profile/
uploadPage");
    modelAndView.addObject("error", messageSource.getMessage("upload.
file.too.big", null, locale));
    return modelAndView;
}
```

Here are the English messages:

```
upload.io.exception=An error occurred while uploading the file. Please
try again.
upload.file.too.big=Your file is too big.
```

Now, the French ones:

```
upload.io.exception=Une erreur est survenue lors de l'envoi du
fichier. Veuillez réessayer.
upload.file.too.big=Votre fichier est trop gros.
```

Placing the profile in a session

The next thing we want is the profile to be stored in a session so that it does not get reset every time we go on the profile page. This can apparently prove tiresome to some users and we have to address it.



HTTP sessions are a way to store information between requests. HTTP is a stateless protocol, which means that there is no way to relate two requests coming from the same user. What most Servlet containers do is they associate a cookie called JSESSIONID to each user. This cookie will be transmitted in the request header and will allow you to store arbitrary objects in a map, an abstraction called HttpSession. Such a session will typically end when the user closes or switches web browsers or after a predefined period of inactivity.

We just saw a method to put objects in a session using the `@SessionAttributes` annotation. This works well within a controller but makes the data difficult to share when spread across multiple controllers. We have to rely on a string to resolve the attribute from its name, which is hard to refactor. For the same reason, we don't want to manipulate the `HttpSession` directly. Another argument that will discourage the direct usage of the session is how difficult it is to unit test the controller that depends on it.

There is another popular approach when it comes to saving things in a session with Spring: annotate a bean with `@Scope("session")`.

You will then be able to inject your session bean in your controllers and other Spring components to either set or retrieve values from it.

Let's create a `UserProfileSession` class in the `profile` package:

```
package masterSpringMvc.profile;

import org.springframework.context.annotation.Scope;
import org.springframework.context.annotation.ScopedProxyMode;
import org.springframework.stereotype.Component;
import java.io.Serializable;
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;

@Component
@Scope(value = "session", proxyMode = ScopedProxyMode.TARGET_CLASS)
public class UserProfileSession implements Serializable {
    private String twitterHandle;
```

File Upload and Error Handling

```
private String email;
private LocalDate birthDate;
private List<String> tastes = new ArrayList<>();

public void saveForm(ProfileForm profileForm) {
    this.twitterHandle = profileForm.getTwitterHandle();
    this.email = profileForm.getEmail();
    this.birthDate = profileForm.getBirthDate();
    this.tastes = profileForm.getTastes();
}

public ProfileForm toForm() {
    ProfileForm profileForm = new ProfileForm();
    profileForm.setTwitterHandle(twitterHandle);
    profileForm.setEmail(email);
    profileForm.setBirthDate(birthDate);
    profileForm.setTastes(tastes);
    return profileForm;
}
}
```

We have conveniently provided a way to convert from and to a `ProfileForm` object. This will help us store and retrieve the form data from our `ProfileController` constructor. We need to inject our `UserProfileSession` variable in the controller's constructor and store it as a field. We also need to expose the `ProfileForm` as a model attribute, which will remove the need to inject it in the `displayProfile` method. Finally, we can save the profile once it has been validated:

```
@Controller
public class ProfileController {

    private UserProfileSession userProfileSession;
    @Autowired
    public ProfileController(UserProfileSession userProfileSession) {
        this.userProfileSession = userProfileSession;
    }

    @ModelAttribute
    public ProfileForm getProfileForm() {
        return userProfileSession.toForm();
    }
}
```

```

    @RequestMapping(value = "/profile", params = {"save"}, method =
RequestMethod.POST)
    public String saveProfile(@Valid ProfileForm profileForm,
BindingResult bindingResult) {
        if (bindingResult.hasErrors()) {
            return "profile/profilePage";
        }
        userProfileSession.saveForm(profileForm);
        return "redirect:/profile";
    }

    // the rest of the code is unchanged
}

```

That's all it takes to save data in a session with Spring MVC.

Now, if you complete the profile form and refresh the page, the data will be persisted between requests.

Just before moving on to the next chapter, I want to detail a couple of concepts we just used.

The first is the injection by the constructor. The `ProfileController` constructor is annotated with `@Autowired`, which means Spring will resolve the constructor arguments from the application context before instantiating the bean. The alternative, which is a little less verbose, would have been to use field injection:

```

@Controller
public class ProfileController {

    @Autowired
    private UserProfileSession userProfileSession;
}

```

Constructor injection is arguably better because it makes the unit testing of our controller easier if we were to move away from the `spring-test` framework and it makes the dependencies of our bean somewhat more explicit.

For a detailed discussion on field injection and constructor injection, refer to the excellent blog post by Oliver Gierke at <http://olivergierke.de/2013/11/why-field-injection-is-evil/>.

Another thing that might need clarification is the `proxyMode` parameter on the `Scope` annotation:

```
@Scope(value = "session", proxyMode = ScopedProxyMode.TARGET_CLASS)
```

There are three `proxyMode` parameters available with Spring, if we don't count the default one:

- `TARGET_CLASS`: This uses a CGI proxy
- `INTERFACES`: This creates a JDK proxy
- `NO`: This does not create any proxy

The advantage of a proxy typically comes into play when you inject something into a long-lived component such as a singleton. Because injection only happens once, when the bean is created, subsequent calls to the injected bean might not reflect its actual state.

In our case, a session bean's actual state is stored in the session and not directly on the bean. This explains why Spring has to create a proxy: it needs to intercept calls to our bean methods and listen for its mutations. This way, the state of the bean can be transparently stored and retrieved from the underlying HTTP session.

For a session bean, we are forced to use a proxy mode. The CGI proxy will instrument your bytecode and work on any class, whereas the JDK approach might be a bit more lightweight but requires you to implement an interface.

Lastly, we made the `UserProfileSession` bean implement the `Serializable` interface. This is not strictly required because the HTTP sessions can store arbitrary objects in memory, but making objects that end up in the session serializable really is a good practice.

Indeed, we might change the way the session is persisted. In fact, we will store the session in a Redis database in *Chapter 8, Optimizing Your Requests*, where Redis has to work with `Serializable` objects. It's always best to think of the session as a generic data store. We have to provide a way to write and read objects from this storage system.

For serialization to work properly on our bean, we also need every one of its field to be `Serializable`. In our case, strings and dates are `Serializable` so we are good to go.

Custom error pages

Spring Boot lets you define your own error view instead of the `Whitelabel` error page that we saw earlier. It must have the name `error` and its purpose is to handle all exceptions. The default `BasicErrorController` class will expose a lot of useful model attributes that you can display on this page.

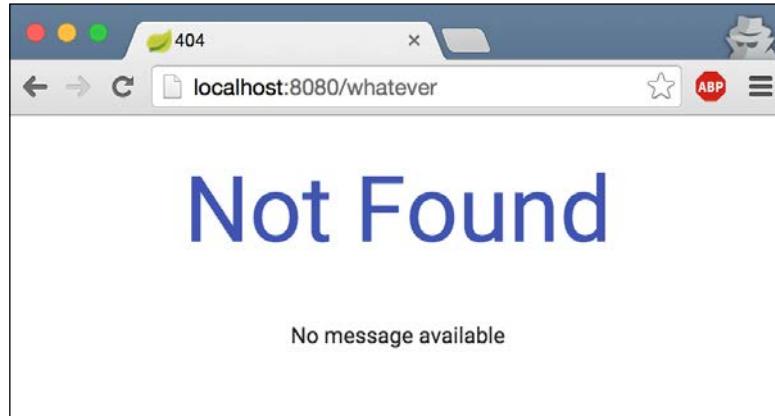
Let's create a custom error page in `src/main/resources/templates`. Let's call it `error.html`:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head lang="en">
    <meta charset="UTF-8"/>
    <title th:text="${status}">404</title>

    <link href="/webjars/materializecss/0.96.0/css/materialize.css"
type="text/css" rel="stylesheet"
media="screen,projection"/>
</head>
<body>
<div class="row">
    <h1 class="indigo-text center" th:text="${error}">Not found</h1>

    <p class="col s12 center" th:text="${message}">
        This page is not available
    </p>
</div>
</body>
</html>
```

Now, if we navigate to a URL that is not handled by our application, we see our custom error page:



A more advanced option to handle errors is to define your own implementation of the `ErrorController` class, a controller in charge of handling all the exceptions at a global level. Take a look at the `ErrorMvcAutoConfiguration` class and the `BasicErrorController` class, which is the default implementation.

URL mapping with matrix variables

We are now aware of what our user is interested in. It would be a good idea to improve our Tweet controller so that it allows searching from a list of keywords.

One interesting way to pass key-value pairs in a URL is to use a matrix variable. It is pretty similar to request parameters. Consider the following code:

```
someUrl/param?var1=value1&var2=value2
```

Instead of the preceding parameter, matrix variables understand this:

```
someUrl/param;var1=value1;var2=value2
```

They also allow each parameter to be a list:

```
someUrl/param;var1=value1,value2;var2=value3,value4
```

A matrix variable can be mapped to different object types inside a controller:

- `Map<String, List<?>>`: This handles multiple variables and multiple values
- `Map<String, ?>`: This handles a case in which each variable has only one value
- `List<?>`: This is used if we are interested in a single variable whose name can be configured

In our case, we want to handle something like this:

```
http://localhost:8080/search/popular;keywords=scala,java
```

The first parameter, `popular`, is the result type known by the Twitter search API. It can take the following values: `mixed`, `recent`, or `popular`.

The rest of our URL is a list of keywords. We will therefore map them to a simple `List<String>` object.

By default, Spring MVC removes every character following a semicolon in a URL. The first thing we need to do to enable matrix variables in our application is to turn off this behavior.

Let's add the following code to our `WebConfiguration` class:

```
@Override
public void configurePathMatch(PathMatchConfigurer configurer) {
    UrlPathHelper urlPathHelper = new UrlPathHelper();
    urlPathHelper.setRemoveSemicolonContent(false);
    configurer.setUrlPathHelper(urlPathHelper);
}
```

Let's create a new controller in the search package, which we will call `SearchController`. Its role is to handle the following request:

```
package masterSpringMvc.search;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.social.twitter.api.Tweet;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.MatrixVariable;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

import java.util.List;

@Controller
public class SearchController {
    private SearchService searchService;
    @Autowired
    public SearchController(SearchService searchService) {
        this.searchService = searchService;
    }

    @RequestMapping("/search/{searchType}")
    public ModelAndView search(@PathVariable String searchType, @
    MatrixVariable List<String> keywords) {
        List<Tweet> tweets = searchService.search(searchType,
        keywords);
        ModelAndView modelAndView = new ModelAndView("resultPage");
        modelAndView.addObject("tweets", tweets);
        modelAndView.addObject("search", String.join(", ", keywords));
        return modelAndView;
    }
}
```

As you can see, we are able reuse the existing result page to display the tweets. We also want to delegate the search to another class called `SearchService`. We will create this service in the same package as `SearchController`:

```
package masterSpringMvc.search;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.social.twitter.api.Tweet;
import org.springframework.social.twitter.api.Twitter;
import org.springframework.stereotype.Service;
```

```
import java.util.List;

@Service
public class SearchService {
    private Twitter twitter;

    @Autowired
    public SearchService(Twitter twitter) {
        this.twitter = twitter;
    }

    public List<Tweet> search(String searchType, List<String>
keywords) {
        return null;
    }
}
```

Now, we need to implement the `search()` method.

The search operation accessible on `twitter.searchOperations().search(params)` takes `searchParameters` as an argument for an advanced search. This object allows us to conduct a search on a dozen of criteria. We are interested in the `query`, `resultType`, and `count` attributes.

First, we need to create a `ResultType` constructor with the `searchType` path variable. The `ResultType` is an enum, so we can iterate over its different values and find one that matches the input, ignoring the case:

```
private SearchParameters.ResultType getResultType(String searchType) {
    for (SearchParameters.ResultType knownType : SearchParameters.
ResultType.values()) {
        if (knownType.name().equalsIgnoreCase(searchType)) {
            return knownType;
        }
    }
    return SearchParameters.ResultType.RECENT;
}
```

We can now create a `SearchParameters` constructor with the following method:

```
private SearchParameters createSearchParam(String searchType, String
taste) {

    SearchParameters.ResultType resultType =
    getResultType(searchType);
```

```
    SearchParameters searchParameters = new SearchParameters(taste);
    searchParameters.resultType(resultType);
    searchParameters.count(3);
    return searchParameters;
}
```

Now, creating a list of the `SearchParameters` constructor is as easy as conducting a map operation (taking a list of keywords and returning a `SearchParameters` constructor for each one):

```
List<SearchParameters> searches = keywords.stream()
    .map(taste -> createSearchParam(searchType, taste))
    .collect(Collectors.toList());
```

Now, we want to fetch the tweets for each `SearchParameters` constructor. You might think of something like this:

```
List<Tweet> tweets = searches.stream()
    .map(params -> twitter.searchOperations().search(params))
    .map(searchResults -> searchResults.getTweets())
    .collect(Collectors.toList());
```

However, if you think about it, this will return a list of tweets. What we want is to flatten all the tweets to get them as a simple list. It turns out that calling `map` and then flattening the result is an operation known as `flatMap`. So we can write:

```
List<Tweet> tweets = searches.stream()
    .map(params -> twitter.searchOperations().search(params))
    .flatMap(searchResults -> searchResults.getTweets().stream())
    .collect(Collectors.toList());
```

The syntax of `flatMap` function, that takes a stream as a parameter, is a bit difficult to understand at first. Let me show you the entire code of the `SearchService` class so we can take a step back:

```
package masterSpringMvc.search;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.social.twitter.api.SearchParameters;
import org.springframework.social.twitter.api.Tweet;
import org.springframework.social.twitter.api.Twitter;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.stream.Collectors;

@Service
```

File Upload and Error Handling

```
public class SearchService {
    private Twitter twitter;

    @Autowired
    public SearchService(Twitter twitter) {
        this.twitter = twitter;
    }

    public List<Tweet> search(String searchType, List<String>
keywords) {
        List<SearchParameters> searches = keywords.stream()
            .map(taste -> createSearchParam(searchType, taste))
            .collect(Collectors.toList());

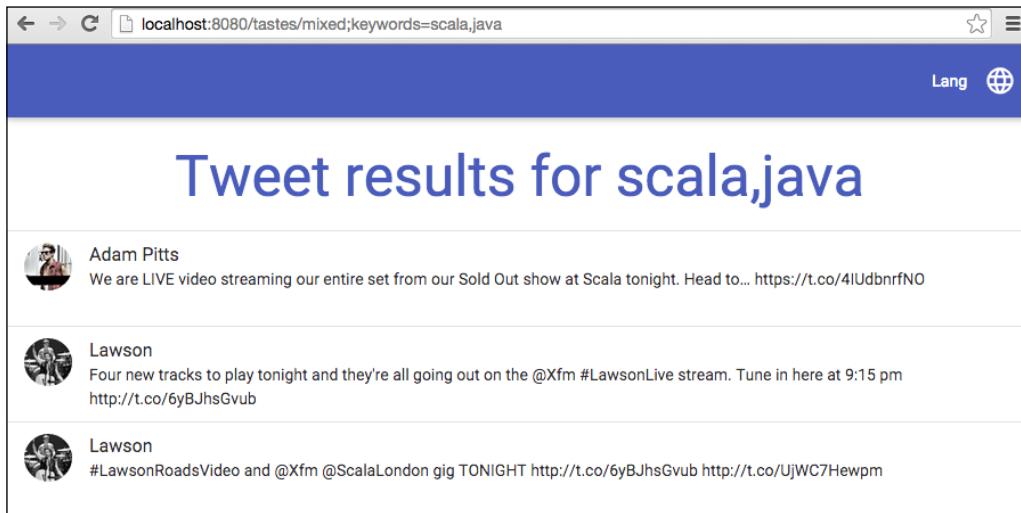
        List<Tweet> results = searches.stream()
            .map(params -> twitter.searchOperations() .
search(params))
            .flatMap(searchResults -> searchResults.getTweets() .
stream())
            .collect(Collectors.toList());

        return results;
    }

    private SearchParameters.ResultType getResultType(String
searchType) {
        for (SearchParameters.ResultType knownType : SearchParameters.
ResultType.values()) {
            if (knownType.name().equalsIgnoreCase(searchType)) {
                return knownType;
            }
        }
        return SearchParameters.ResultType.RECENT;
    }

    private SearchParameters createSearchParam(String searchType,
String taste) {
        SearchParameters.ResultType resultType =
getResultSetType(searchType);
        SearchParameters searchParameters = new
SearchParameters(taste);
        searchParameters.resultType(resultType);
        searchParameters.count(3);
        return searchParameters;
    }
}
```

Now, if we navigate to `http://localhost:8080/search/mixed;keywords=scala,java`, we get the expected result. A search for the Scala keyword and then for Java:



Putting it together

Now that everything works separately, it's time to assemble everything. We will do this in three steps:

1. Move the upload form to the profile page and remove the old upload page.
2. Change the submit button on the profile page to trigger the taste search directly.
3. Change the home page of our application. It should display search results matching our users' tastes right away. If they are unavailable, go to the profile page.

I encourage you to try to do it on your own. You will run into very manageable problems along the way but you should know enough to resolve them on your own. I believe in you.

OK, now that you have done the work (you have, haven't you?), let's take a look at my solution.

The first step is to remove the old `uploadPage` title. Don't look back, just do it.

Next, put these lines just below the `profilePage` title:

```
<div class="row">

    <div class="col m8 s12 offset-m2">
        
    </div>

    <div class="col s12 center red-text" th:text="${error}"
        th:if="${error}">
        Error during upload
    </div>

    <form th:action="@{/profile}" method="post" enctype="multipart/
    form-data" class="col m8 s12 offset-m2">

        <div class="input-field col s6">
            <input type="file" id="file" name="file"/>
        </div>

        <div class="col s6 center">
            <button class="btn indigo waves-effect waves-light"
                type="submit" name="upload" th:text="#{upload}">Upload
                <i class="mdi-content-send right"></i>
            </button>
        </div>
    </form>
</div>
```

It is very similar to the content of the late `uploadPage`. We just removed the title and changed the label of the submit button. Add the corresponding translation to the bundles.

In English:

upload=Upload

In French:

Upload=Envoyer

We also changed the name of the submit button to `upload`. That will help us identify this action on the controller side.

Now, if we try to upload our picture, it will redirect us to the old upload page. We need to fix this in the `onUpload` method of our `PictureUploadController` class:

```

@RequestMapping(value = "/profile", params = {"upload"}, method =
RequestMethod.POST)
public String onUpload(@RequestParam MultipartFile file,
RedirectAttributes redirectAttrs) throws IOException {

    if (file.isEmpty() || !isImage(file)) {
        redirectAttrs.addFlashAttribute("error", "Incorrect file.
Please upload a picture.");
        return "redirect:/profile";
    }

    Resource picturePath = copyFileToPictures(file);
    userProfileSession.setPicturePath(picturePath);

    return "redirect:profile";
}

```

Note that we changed the URL that handles the post. It is now `/profile` instead of `/upload`. Form handling is much simpler when the `GET` and `POST` requests have the same URL, and will save us a lot of trouble especially when dealing with exceptions. This way, we will not have to redirect the user after an error.

We also removed the model attribute, `picturePath`. Since we now have a bean representing our user in a session, `UserProfileSession`, we decided to add it there. We added a `picturePath` attribute to the `UserProfileSession` class and the associated getters and setters.

Don't forget to inject the `UserProfileSession` class and make it available as a field in our `PictureUploadController` class.

Remember that all the properties of our session bean must be serializable, unlike resources. So we need to store it differently. The `URL` class seems to be a good fit. It is serializable and it is easy to create a resource from a `URL` with the `UrlResource` class:

```

@Component
@Scope(value = "session", proxyMode = ScopedProxyMode.TARGET_CLASS)
public class UserProfileSession implements Serializable {
    private URL picturePath;

```

File Upload and Error Handling

```
    public void setPicturePath(Resource picturePath) throws
    IOException {
        this.picturePath = picturePath.getURL();
    }

    public Resource getPicturePath() {
        return picturePath == null ? null : new
        UrlResource(picturePath);
    }
}
```

The last thing that I had to do is to make the `profileForm` available as a model attribute after an error. This is because the `profilePage` requires it when it is rendered.

To sum up, here is the final version of the `PictureUploadController` class:

```
package masterSpringMvc.profile;

import masterSpringMvc.config.PictureUploadProperties;
import org.apache.tomcat.util.http.fileupload.IOUtils;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.MessageSource;
import org.springframework.core.io.FileSystemResource;
import org.springframework.core.io.Resource;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.multipart.MultipartFile;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.support.RedirectAttributes;

import javax.servlet.http.HttpServletResponse;
import java.io.*;
import java.net.URLConnection;
import java.util.Locale;

@Controller
public class PictureUploadController {
    private final Resource picturesDir;
    private final Resource anonymousPicture;
    private final MessageSource messageSource;
    private final UserProfileSession userProfileSession;
```

```

    @Autowired
    public PictureUploadController(PictureUploadProperties
        uploadProperties,
                                         MessageSource messageSource,
                                         UserProfileSession
        userProfileSession) {
        picturesDir = uploadProperties.getUploadPath();
        anonymousPicture = uploadProperties.getAnonymousPicture();
        this.messageSource = messageSource;
        this.userProfileSession = userProfileSession;
    }

    @RequestMapping(value = "/uploadedPicture")
    public void getUploadedPicture(HttpServletRequest response)
        throws IOException {
        Resource picturePath = userProfileSession.getPicturePath();
        if (picturePath == null) {
            picturePath = anonymousPicture;
        }
        response.setHeader("Content-Type", URLConnection.guessContentTypeFromName(picturePath.getFilename()));
        IOUtils.copy(picturePath.getInputStream(), response.getOutputStream());
    }

    @RequestMapping(value = "/profile", params = {"upload"}, method =
        RequestMethod.POST)
    public String onUpload(@RequestParam MultipartFile file,
        RedirectAttributes redirectAttrs) throws IOException {

        if (file.isEmpty() || !isImage(file)) {
            redirectAttrs.addFlashAttribute("error", "Incorrect file.
Please upload a picture.");
            return "redirect:/profile";
        }

        Resource picturePath = copyFileToPictures(file);
        userProfileSession.setPicturePath(picturePath);

        return "redirect:profile";
    }

    private Resource copyFileToPictures(MultipartFile file) throws
        IOException {

```

File Upload and Error Handling

```
        String fileExtension = getFileExtension(file.
getOriginalFilename());
        File tempFile = File.createTempFile("pic", fileExtension,
picturesDir.getFile());
        try (InputStream in = file.getInputStream();
            OutputStream out = new FileOutputStream(tempFile)) {

            IOUtils.copy(in, out);
        }
        return new FileSystemResource(tempFile);
    }

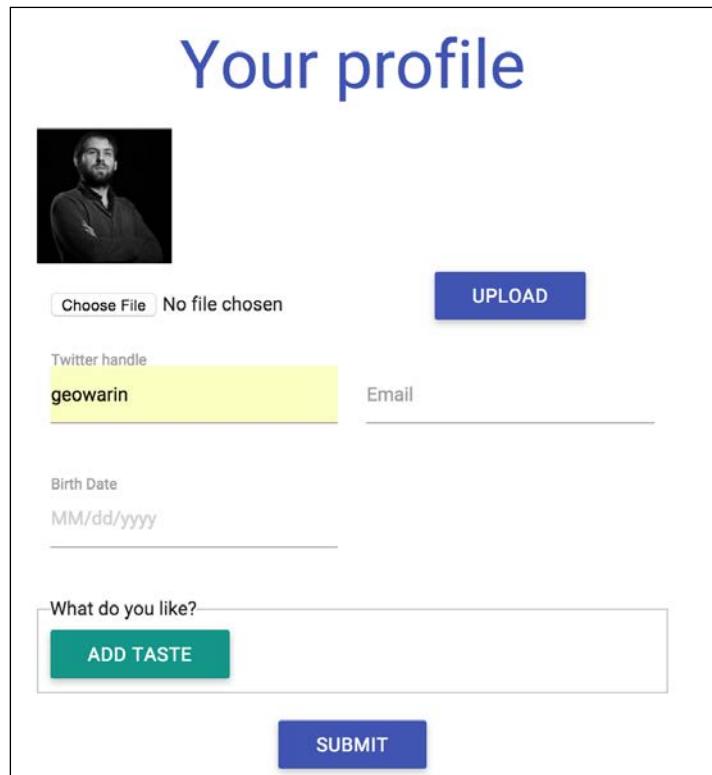
    @ExceptionHandler(IOException.class)
    public ModelAndView handleIOException(Locale locale) {
        ModelAndView modelAndView = new ModelAndView("profile/
profilePage");
        modelAndView.addObject("error", messageSource.
getMessage("upload.io.exception", null, locale));
        modelAndView.addObject("profileForm", userProfileSession.
toForm());
        return modelAndView;
    }

    @RequestMapping("uploadError")
    public ModelAndView onUploadError(Locale locale) {
        ModelAndView modelAndView = new ModelAndView("profile/
profilePage");
        modelAndView.addObject("error", messageSource.
getMessage("upload.file.too.big", null, locale));
        modelAndView.addObject("profileForm", userProfileSession.
toForm());
        return modelAndView;
    }

    private boolean isImage(MultipartFile file) {
        return file.getContentType().startsWith("image");
    }

    private static String getFileExtension(String name) {
        return name.substring(name.lastIndexOf("."));
    }
}
```

So, now we can go to the profile page and upload our picture as well as provide personal information, as shown in the following screenshot:



Now, let's redirect our user to its search after the profile is completed. For this, we need to modify the `saveProfile` method in the `ProfileController` class:

```
@RequestMapping(value = "/profile", params = {"save"}, method = RequestMethod.POST)
public String saveProfile(@Valid ProfileForm profileForm,
BindingResult bindingResult) {
    if (bindingResult.hasErrors()) {
        return "profile/profilePage";
    }
    userProfileSession.saveForm(profileForm);
    return "redirect:/search/mixed;keywords=" + String.join(",",
profileForm.getTastes());
}
```

Now that we are able to search for tweets from our profile, we don't need the `searchPage` or `TweetController` we previously made. Simply delete the `searchPage.html` page and the `TweetController`.

To finish, we can modify our home page so that it redirects us to a search matching our tastes if we have already completed our profile.

Let's create a new controller in the controller package. It is responsible for redirecting a user arriving at the root of our website either to their profile if it's incomplete or to the `resultPage` if their tastes are available:

```
package masterSpringMvc.controller;

import masterSpringMvc.profile.UserProfileSession;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

import java.util.List;

@Controller
public class HomeController {
    private UserProfileSession userProfileSession;

    @Autowired
    public HomeController(UserProfileSession userProfileSession) {
        this.userProfileSession = userProfileSession;
    }

    @RequestMapping("/")
    public String home() {
        List<String> tastes = userProfileSession.getTastes();
        if (tastes.isEmpty()) {
            return "redirect:/profile";
        }
        return "redirect:/search/mixed;keywords=" + String.join(",",
        tastes);
    }
}
```

The check point

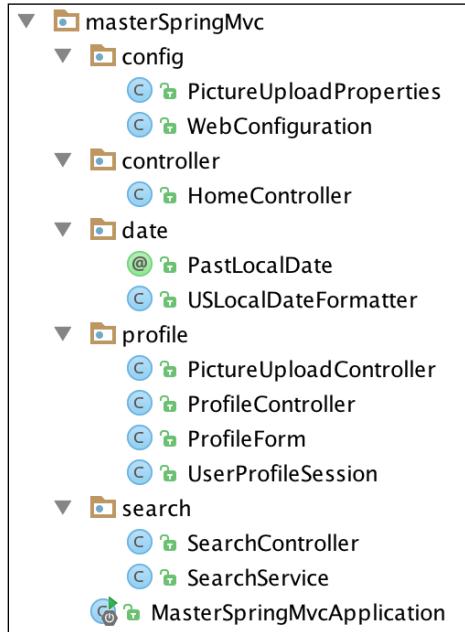
In this chapter, we added two controllers, the `PictureUploadController`, which is in charge of writing uploaded files to the disk and handling upload errors, and the `SearchController` that can search tweets from a list of keywords with matrix parameters.

This controller then delegates the search to a new service, `SearchService`.

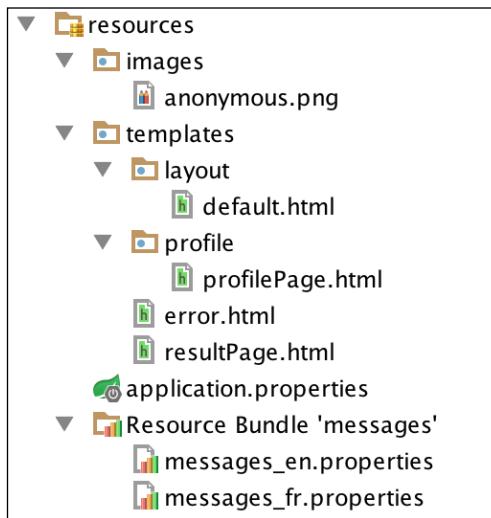
We deleted the old `TweetController`.

We created a session bean, `UserProfileSession`, to store the information about our user.

Finally, we added two things to `WebConfiguration`. We added the error pages for our Servlet container and support for matrix variables.



On the resources side, we added a picture representing an anonymous user and a static page to handle errors. We added the file upload to `profilePage` and got rid of the old `searchPage`.



Summary

In this chapter, we discussed file upload and error handling. Uploading a file is not really complicated. However, a big design decision is what to do with the uploaded files. We could have stored the images in a database, but instead we chose to write it to the disk and save the location of each user's picture in their session.

We saw typical ways to handle exceptions at the controller level and at the servlet container level. For additional resources on Spring MVC error handling, you can refer to the blog post at <https://spring.io/blog/2013/11/01/exception-handling-in-spring-mvc>.

Our application is looking pretty good already and yet the amount of code we had to write is very reasonable.

Stay tuned for the next chapter where we will see that Spring MVC is also a powerful framework to build REST applications.

5

Crafting a RESTful Application

In this chapter, we will have a look at the main principles of a RESTful architecture. Then, with the help of very handy tools, we will design a friendly API, leveraging Jackson's capabilities to serialize our model in JSON.

We will document our application with the appropriate error codes and HTTP verbs and automatically generate a neat frontend for our application by using Swagger UI.

Finally, we will look at the other forms of serialization and learn more about the content negotiation mechanism of Spring MVC.

What is REST?

REST (Representational State Transfer) is an architectural style that defines best practices for creating scalable web services leveraging the capabilities of the HTTP protocol.

A RESTful web service should naturally exhibit the following properties:

- **Client-server:** The UI is separated from data storage
- **Stateless:** Each request contains enough information for the server to operate without maintaining any state
- **Cacheable:** The server's responses contain enough information to allow the clients to make sensible decisions about data storage
- **Uniform interface:** URIs uniquely identify resources and hyperlinks allow the API to be discovered
- **Layered:** Each resource of the API provides a sensible level of detail

The advantage of such an architecture is that it is simple to maintain and easy to discover. It also scales well because there is no need to maintain a persistent connection between the server and the client, which eliminates the need for load balancing or sticky sessions. Finally, the service is more efficient because the information is neatly laid out and easy to cache.

Let's see how we can design better APIs incrementally by using Richardson's maturity model.

Richardson's maturity model

Leonard Richardson is famous for having defined four levels, ranked from 0 to 3, that describe the level of "RESTfulness" of a web API. Each level requires additional work and investment in the API but also provides additional benefits.

Level 0 – HTTP

Level 0 is really easy to reach; you just have to make your resource available on a network through the HTTP protocol. You can use any data representation you find best suited for your use case (XML, JSON, and so on).

Level 1 – Resources

Most people think of resources when they hear the term REST. A resource is a unique identifier for an element of our model, a user or a tweet, for instance. With HTTP, a resource is obviously associated with a unified resource identifier URI, as shown in this example:

- `/users` contains the list of all our users
- `/user/42` contains a specific user
- `/user/42/tweets` contains the list of all the tweets associated to this particular user

Maybe your API could allow access to a particular tweet related to a user with `/user/42/tweet/3` or maybe each tweet is uniquely identified, in which case you might prefer `/tweet/3`.

The goal of this level is to deal with the complexity of an application by exposing multiple specialized resources.

There is no rule regarding the type of response that your server can return. You might want to include only scarce information when you list all the resources with `/users` and give more details when a specific resource is requested. Some APIs even let you list the fields you are interested in before serving them to you.

It really is up to you to define the form of your API, keeping one simple rule in mind: the principle of least astonishment. Give your users what they expect and your API will already be in good shape.

Level 2 – HTTP verbs

This level is about using the HTTP verbs to identify possible actions on the resources. This is a very good way to describe what can be done with your API since the HTTP verbs are a well-known standard among developers.

The main verbs are listed here:

- `GET`: This reads data on a particular URI.
- `HEAD`: This does the same as `GET` without the response body. This is useful for getting metadata on a resource (cache information and so on).
- `DELETE`: This deletes a resource.
- `PUT`: This updates or creates a resource.
- `POST`: This updates or creates a resource.
- `PATCH`: This partially updates a resource.
- `OPTIONS`: This returns the list of methods that the server supports on a particular resource.

Most applications that allow **Create Read Update Delete (CRUD)** operations get by with only three verbs: `GET`, `DELETE`, and `POST`. The more verbs you implement, the richer and more semantic your API becomes. It helps third parties to interact with your service by allowing them to type a few commands and see what happens.

The `OPTIONS` and `HEAD` verbs are rarely seen because they work on the metadata level and are typically not vital to any application.

At first sight, the `PUT` and `POST` verbs appear to do the same thing. The main difference is that the `PUT` verb is said to be idempotent, which means that sending the same request multiple times should result in the same server state. The implication of that rule is essentially that the `PUT` verb should operate on a given URI and contain enough information for the request to succeed.

For instance, a client can use `PUT` data on `/user/42`, and the result will be either an update or a creation, depending on whether the entity existed prior to the request.

On the other hand, `POST` should be used when you don't exactly know what URI you should write to. You could send `POST` to `/users` without specifying an ID in the request and expect the user to be created. You could also send `POST` to the same `/users` resource, this time specifying a user ID inside the request entity and expect the server to update the corresponding user.

As you can see, both of these options work. One frequent use case is to use `POST` for creation (because, most of the time, the server should be in charge of the IDs) and to use `PUT` to update a resource whose ID is already known.

The server might also allow a resource to be modified partially (without the client sending the full contents of the resource). It should respond to the `PATCH` method in that case.

At this level, I also encourage you to use meaningful HTTP codes when providing responses. We will see the most common codes in a moment.

Level 3 – Hypermedia controls

Hypermedia controls are also known as **Hypertext As The Engine Of Application State (HATEOAS)**. Behind this barbarous acronym lies the most important property of a RESTful service: making it discoverable through the use of hypertext links. This is essentially the server telling the client what its options are, using the response headers or the response entity.

For instance, after the creation of a resource with `PUT`, the server should return a response with the code `201 CREATED` and send a `Location` header containing the URI of the created resource.

There is no standard that defines how the link to the other parts of the API should look. Spring Data REST, a Spring project that allows you to create a RESTful backend with minimal configuration, typically outputs this:

```
{  
  "_links" : {  
    "people" : {  
      "href" : "http://localhost:8080/users{?page,size,sort}",  
      "templated" : true  
    }  
  }  
}
```

Then, go to /users:

```
{  
  "_links" : {  
    "self" : {  
      "href" : "http://localhost:8080/users{?page,size,sort}",  
      "templated" : true  
    },  
    "search" : {  
      "href" : "http://localhost:8080/users/search"  
    }  
  "page" : {  
    "size" : 20,  
    "totalElements" : 0,  
    "totalPages" : 0,  
    "number" : 0  
}
```

This gives you a good idea of what you can do with the API, doesn't it?

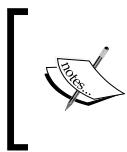
API versioning

If third-party clients use your API, you could consider versioning your API to avoid breaking changes when you update your application.

Versioning an API is often a matter of making a set of stable resources available under subdomains. For instance, GitLab maintains three versions of its API. They are accessible under <https://example/api/v3>, and so on. Like a lot of architectural decisions in software, versioning is a tradeoff.

It will require more work to design such an API and identify breaking changes in the API. Often, the addition of new fields will not be as problematic as removing or transforming the API entity results or requests.

Most of the time, you will be in charge of both the API and the client, thereby removing the need for such sophistication.



See this blog post for a more in-depth discussion about API versioning:
<http://www.troyhunt.com/2014/02/your-api-versioning-is-wrong-which-is.html>

Useful HTTP codes

Another important aspect of a good RESTful API is to use HTTP codes in a sensible way. The HTTP specification defines a lot of standard codes. They should cover 99 percent of what a good API needs to communicate to its users. The following list contains the most important codes, the ones every API should use and every developer should know:

Code	Meaning	Usage
2xx - Success	These codes are used when everything goes well.	
200	Everything is okay.	The request succeeded.
201	A resource has been created.	The successful creation of a resource. The response should include a list of locations associated with the creation.
204	There is no content to return.	The server has successfully handled the request but there is no content to return.
3xx - Redirection	These codes are used when further action is needed on the client to fulfill the request.	
301	Moved permanently	The resource has a changed URI and its new location is indicated in the <code>Location</code> header.
304	The resource has not been modified.	The resource has not changed since the last time. This response must include the date, <code>ETag</code> , and cache information.

Code	Meaning	Usage
4xx - Client error	The request was not successfully performed because of a mistake made by the client.	
400	Bad request	The data sent by the client could not be understood.
403	Forbidden	The request was understood but not allowed. This can be enriched with information describing the error.
404	Not found	Nothing matches this URI. This can be used instead of 403 if information about security shouldn't be disclosed.
409	Conflict	The request conflicts with another modification. The response should include information on how to resolve the conflict.
5xx - Server error	An error occurred on the server side.	
500	An internal server error	The server unexpectedly failed to process the request.



For a more detailed list, see <http://www.restapitutorial.com/httpstatuscodes.html>.

Client is the king

We will allow third-party clients to retrieve the search results via a REST API. These results will be available either in JSON or XML.

We want to handle requests of the `/api/search/mixed;keywords=springFramework` form. This is really similar to the search form we already made, except that the request path begins with `api`. Every URI found in this namespace should return binary results.

Let's create a new `SearchApiController` class in the `search.api` package:

```
package masterSpringMvc.search.api;

import masterSpringMvc.search.SearchService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.social.twitter.api.Tweet;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/search")
public class SearchApiController {
    private SearchService searchService;

    @Autowired
    public SearchApiController(SearchService searchService) {
        this.searchService = searchService;
    }

    @RequestMapping(value =("/{searchType}", method = RequestMethod.
    GET)
    public List<Tweet> search(@PathVariable String searchType, @
    MatrixVariable List<String> keywords) {
        return searchService.search(searchType, keywords);
    }
}
```

This is quite similar to our previous controller, with three subtle differences:

- The controller class is annotated with a `@RequestMapping` annotation. This will be our base address and will prefix every other mapping declared in this controller.
 - We no longer redirect to a view but return a plain object in the search method.
 - The controller is annotated with `@RestController` instead of `@Controller`.

The `RestController` is a shortcut to declare controllers that will return each response as if it were annotated with the `@ResponseBody` annotation. It tells Spring to serialize the return type to the appropriate format, which is JSON by default.

When working with a REST API, a good practice is to always specify the method you will respond to. It's rather unlikely that a request can be handled the same way for a GET or a POST method.

If you go to `http://localhost:8080/api/search/mixed;keywords=springFramework`, you should get a really large result, as follows:

```
localhost:8080/api/search/mixed;keywords=springFramework
Raw Parsed

{
  "extraData": {},
  "id": 629241388719304700,
  "text": "DHL Capability Tool: http://t.co/3oAE4UzPuh.client.WebServiceTransportException: Internal Server Error [500]! \nWhat's a shame DHL!",
  "createdAt": 1438857812000,
  "fromUser": "sina689",
  "profileImageURL": "http://pbs.twimg.com/profile_images/526127140337512448/vBEEZfQ5_normal.jpeg",
  "toUserId": 0,
  "inReplyToStatusId": null,
  "inReplyToUserId": null,
  "inReplyToScreenName": "null",
  "fromUserId": 629364501,
  "languageCode": "en",
  "source": "<a href=\"http://twitter.com\" rel=\"nofollow\">Twitter Web Client</a>",
  "retweetCount": 0,
  "retweeted": false,
  "retweetedStatus": null,
  "favorited": false,
  "favoriteCount": 0,
}
```

Indeed, Spring handled the serialization of the whole `Tweet` class' attributes automatically, using Jackson.

Debugging a RESTful API

With your browser, you will only be able to perform GET requests on a specific API. The good tools will make your developments much simpler. There are lots of tools to test a RESTful API. I will just list the one I use and love.

A JSON formatting extension

Often, you will just test the GET method and your first reflex will be to copy the address into your browser to check the result. In that case, you have the possibility to get more than plain text with extensions such as JSON Formatter for Chrome or JSONView for Firefox.

A RESTful client in your browser

The browser is the natural tool for dealing with HTTP requests. However, using the address bar will rarely allow you to test your API in detail.

Postman is an extension for Chrome, and RESTClient is its Firefox counterpart. They both have similar features, such as creating and sharing collections of queries, modification of the headers, and handling authentication (basic, digest, and OAuth). At the time of writing, only RESTClient handles OAuth2.

httpie

httpie is a command line utility à la curl but oriented towards REST querying. It allows you to type commands such as this:

```
http PUT httpbin.org/put hello=world
```

It's a lot friendlier than this ugly version:

```
curl -i -X PUT httpbin.org/put -H Content-Type:application/json -d
'{"hello": "world"}'
```

Customizing the JSON output

Using our tools we are able to easily see the request generated by our server. It is huge. By default, Jackson, the JSON serialization library used by Spring Boot, will serialize everything that is accessible with a getter method.

We would like something lighter, such as this:

```
{  
  "text": "original text",  
  "user": "some_dude",  
  "profileImageUrl": "url",  
  "lang": "en",  
  "date": 2015-04-15T20:18:55,  
  "retweetCount": 42  
}
```

The easiest way to customize which fields will be serialized is by adding annotations to our beans. You can either use the `@JsonIgnoreProperties` annotation at the class level to ignore a set of properties or add `@JsonIgnore` on the getters of the properties you wish to ignore.

In our case, the `Tweet` class is not one of our own. It is part of Spring Social Twitter, and we do not have the ability to annotate it.

Using the model classes directly for serialization is rarely a good option. It would tie your model to your serialization library, which should remain an implementation detail.

When dealing with unmodifiable code, Jackson provides two options:

- Creating a new class dedicated to serialization.
- Using mixins, which are simple classes that will be linked to your model. These will be declared in your code and can be annotated with any Jackson annotation.

Since we only need to perform some simple transformation on the fields of our model (a lot of hiding and a little renaming), we could opt for the mixins.

It's a good, non-invasive way to rename and exclude fields on the fly with a simple class or interface.

Another option to specify subsets of fields used in different parts of the application is to annotate them with the `@JsonView` annotation. This won't be covered in this chapter, but I encourage you to check out this excellent blog post <https://spring.io/blog/2014/12/02/latest-jackson-integration-improvements-in-spring>.

We want to be in control of the output of our APIs, so let's just create a new class called `LightTweet` that can be constructed from a tweet:

```
package masterSpringMvc.search;

import org.springframework.social.twitter.api.Tweet;
import org.springframework.social.twitter.api.TwitterProfile;

import java.time.LocalDateTime;
import java.time.ZoneId;
import java.util.Date;

public class LightTweet {
    private String profileImageUrl;
    private String user;
    private String text;
    private LocalDateTime date;
    private String lang;
    private Integer retweetCount;

    public LightTweet(String text) {
        this.text = text;
    }

    public static LightTweet ofTweet(Tweet tweet) {
        LightTweet lightTweet = new LightTweet(tweet.getText());
        Date createdAt = tweet.getCreatedAt();
        if (createdAt != null) {
            lightTweet.date = LocalDateTime.ofInstant(createdAt.
toInstant(), ZoneId.systemDefault());
        }
        TwitterProfile tweetUser = tweet.getUser();
        if (tweetUser != null) {
            lightTweet.user = tweetUser.getName();
            lightTweet.profileImageUrl = tweetUser.
getProfileImageUrl();
        }
    }
}
```

```

        lightTweet.lang = tweet.getLanguageCode();
        lightTweet.retweetCount = tweet.getRetweetCount();
        return lightTweet;
    }

    // don't forget to generate getters
    // They are used by Jackson to serialize objects
}

```

We now need to make our `SearchService` class return the `LightTweets` class instead of tweets:

```

public List<LightTweet> search(String searchType, List<String>
keywords) {
    List<SearchParameters> searches = keywords.stream()
        .map(taste -> createSearchParam(searchType, taste))
        .collect(Collectors.toList());

    List<LightTweet> results = searches.stream()
        .map(params -> twitter.searchOperations().
    search(params))
        .flatMap(searchResults -> searchResults.getTweets() .
    stream())
        .map(LightTweet::ofTweet)
        .collect(Collectors.toList());

    return results;
}

```

This will impact the return type of the `SearchApiController` class as well as the `tweets` model attribute in the `SearchController` class. Make the necessary modification in those two classes.

We also need to change the code of the `resultPage.html` file because some properties changed (we no longer have a nested `user` property):

```

<ul class="collection">
    <li class="collection-item avatar" th:each="tweet : ${tweets}">
        
        <span class="title" th:text="${tweet.user}">Username</span>

        <p th:text="${tweet.text}">Tweet message</p>
    </li>
</ul>

```

We're almost done. If you restart your application and go to <http://localhost:8080/api/search/mixed;keywords=springFramework>, you'll see that the date format is not the one we expected:



```
[{"profileImageUrl": "http://pbs.twimg.com/profile_images/471877622436069376rma1.jpeg", "user": "Cesar A. Nogueira", "text": "RT @SiliconArmada: (Sr.) Developer Hybris Job @SAP #Kiev http://t.co/BgQ53dQQJY #Java #SpringFramework #JEE #hybris #CRM", "date": {"hour": 15, "minute": 2, "second": 40, "nano": 0, "year": 2015, "month": "AUGUST", "dayOfMonth": 2, "dayOfWeek": "SUNDAY", "dayOfYear": 214, "monthValue": 8, "chronology": {"id": "ISO", "calendarType": "iso8601"}}, "lang": "fr", "retweetCount": 4},
```

That's because Jackson doesn't have built-in support for JSR-310 dates. Luckily, this is easy to fix. Simply add the following library to the dependencies in the `build.gradle` file:

```
compile 'com.fasterxml.jackson.datatype:jackson-datatype-jsr310'
```

This indeed changes the date format, but it now outputs an array instead of a formatted date.

To change that, we need to understand what the library did. It includes a new Jackson module called JSR-310 Module. A Jackson module is an extension point to customize serialization and deserialization. This one will automatically be registered by Spring Boot at startup in the `JacksonAutoConfiguration` class, which will create a default Jackson `ObjectMapper` method with support for well-known modules.

We can see that the former module adds a bunch of serializers and deserializers for all the new classes defined in JSR-310. This will try to convert every date to an ISO format, whenever possible. See <https://github.com/FasterXML/jackson-datatype-jsr310>.

If we take a closer look at `LocalDateTimeSerializer`, for instance, we can see that it actually has two modes and can switch between the two with a serialization feature called `WRITE_DATES_AS_TIMESTAMPS`.

To define this property, we need to customize Spring's default object mapper. As we can gather from looking at the auto configuration, Spring MVC provides a utility class to create the `ObjectMapper` method that we can use. Add the following bean to your `WebConfiguration` class:

```
@Bean
@Primary
public ObjectMapper objectMapper(Jackson2ObjectMapperBuilder builder)
{
    ObjectMapper objectMapper = builder.createXmlMapper(false).build();
    objectMapper.configure(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS, false);
    return objectMapper;
}
```

This time, we are done and the dates are properly formatted, as you can see here:



A user management API

Our search API is quite good, but let's do something more interesting. Like a lot of web applications, we will need a user management module to identify our users. For that, we will create a new user package. In this package, we will add a model class as follows:

```
package masterSpringMvc.user;

import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;

public class User {
    private String twitterHandle;
    private String email;
    private LocalDate birthDate;
    private List<String> tastes = new ArrayList<>();

    // Getters and setters for all fields
}
```

Since we do not want to use a database just yet, we will create a `UserRepository` class in the same package, backed by a simple `Map`:

```
package masterSpringMvc.user;

import org.springframework.stereotype.Repository;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

@Repository
public class UserRepository {
    private final Map<String, User> userMap = new
    ConcurrentHashMap<>();

    public User save(String email, User user) {
        user.setEmail(email);
        return userMap.put(email, user);
    }
}
```

```
public User save(User user) {
    return save(user.getEmail(), user);
}

public User findOne(String email) {
    return userMap.get(email);
}

public List<User> findAll() {
    return new ArrayList<>(userMap.values());
}

public void delete(String email) {
    userMap.remove(email);
}

public boolean exists(String email) {
    return userMap.containsKey(email);
}
```

Finally, in the `user.api` package, we will create a very naive controller implementation:

```
package masterSpringMvc.user.api;

import masterSpringMvc.user.User;
import masterSpringMvc.user.UserRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api")
public class UserApiController {

    private UserRepository userRepository;

    @Autowired
    public UserApiController(UserRepository userRepository) {
        this.userRepository = userRepository;
    }
```

```
    @RequestMapping(value = "/users", method = RequestMethod.GET)
    public List<User> findAll() {
        return userRepository.findAll();
    }

    @RequestMapping(value = "/users", method = RequestMethod.POST)
    public User createUser(@RequestBody User user) {
        return userRepository.save(user);
    }

    @RequestMapping(value = "/user/{email}", method = RequestMethod.
PUT)
    public User updateUser(@PathVariable String email, @RequestBody
User user) {
        return userRepository.save(email, user);
    }

    @RequestMapping(value = "/user/{email}", method = RequestMethod.
DELETE)
    public void deleteUser(@PathVariable String email) {
        userRepository.delete(email);
    }
}
```

We implemented all the classic CRUD operations with a RESTful repository by using the user's e-mail address as a unique identifier.

In this scenario, you will quickly face problems as Spring strips contents found after a dot. The solution is very similar to what we use to support semicolons in URLs in the URL mapping with matrix variables section in *Chapter 4, File Upload and Error Handling*.

Add the `useRegisteredSuffixPatternMatch` property that is set to false in the `configurePathMatch()` method that we have already defined in the `WebConfiguration` class:

```
    @Override
    public void configurePathMatch(PathMatchConfigurer configurer) {
        UrlPathHelper urlPathHelper = new UrlPathHelper();
        urlPathHelper.setRemoveSemicolonContent(false);
        configurer.setUrlPathHelper(urlPathHelper);
        configurer.setUseRegisteredSuffixPatternMatch(true);
    }
```

Now that we've got our API, we can start interacting with it.

Here are a few sample commands with httpie:

```
~ $ http get http://localhost:8080/api/users
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Date: Mon, 20 Apr 2015 00:01:08 GMT
Server: Apache-Coyote/1.1
Transfer-Encoding: chunked

[]

~ $ http post http://localhost:8080/api/users email=geo@springmvc.com
birthDate=2011-12-12 tastes='["spring"]'
HTTP/1.1 200 OK
Content-Length: 0
Date: Mon, 20 Apr 2015 00:02:07 GMT
Server: Apache-Coyote/1.1

~ $ http get http://localhost:8080/api/users
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Date: Mon, 20 Apr 2015 00:02:13 GMT
Server: Apache-Coyote/1.1
Transfer-Encoding: chunked

[
  {
    "birthDate": "2011-12-12",
    "email": "geo@springmvc.com",
    "tastes": [
      "spring"
    ],
    "twitterHandle": null
  }
]
```

1

```
~ $ http delete http://localhost:8080/api/user/geo@springmvc.com
HTTP/1.1 200 OK
Content-Length: 0
Date: Mon, 20 Apr 2015 00:02:42 GMT
Server: Apache-Coyote/1.1
```

```
~ $ http get http://localhost:8080/api/users
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Date: Mon, 20 Apr 2015 00:02:46 GMT
Server: Apache-Coyote/1.1
Transfer-Encoding: chunked
```

[]

This is good but not great. Status codes are not yet handled. We will need more RESTfulness to climb up the Richardson ladder.

Status codes and exception handling

The first thing we want to do is to correctly handle response statuses. By default, Spring automatically deals with some statuses:

- 500 Server Error: This indicates that an exception occurred while handling the request.
- 405 Method not Supported: This comes up when you use an incorrect method on an existing handler.
- 404 Not Found: This comes up when the handler does not exist.
- 400 Bad Request: This indicates that the request body or parameter does not match the server's expectation.
- 200 OK: It is thrown for any request handled without an error.

With Spring MVC, there are two ways to return status codes:

- Returning a `ResponseEntity` class from a REST controller
- Throwing an exception that will be caught in dedicated handlers

Status code with `ResponseEntity`

The HTTP protocol specifies that we should return a `201 Created` status when we create a new user. With our API, this can happen with a `POST` method. We also need to throw some `404` errors on operation while working on an entity that does not exist.

Spring MVC has a class that associates an HTTP status with a response entity. It is called `ResponseEntity`. Let's update our `UserApiController` class to handle error codes:

```
package masterSpringMvc.user.api;

import masterSpringMvc.user.User;
import masterSpringMvc.user.UserRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api")
public class UserApiController {

    private UserRepository userRepository;

    @Autowired
    public UserApiController(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    @RequestMapping(value = "/users", method = RequestMethod.GET)
    public List<User> findAll() {
        return userRepository.findAll();
    }
}
```

```
    @RequestMapping(value = "/users", method = RequestMethod.POST)
    public ResponseEntity<User> createUser(@RequestBody User user) {
        HttpStatus status = HttpStatus.OK;
        if (!userRepository.exists(user.getEmail())) {
            status = HttpStatus.CREATED;
        }
        User saved = userRepository.save(user);
        return new ResponseEntity<>(saved, status);
    }

    @RequestMapping(value = "/user/{email}", method = RequestMethod.PUT)
    public ResponseEntity<User> updateUser(@PathVariable String email,
    @RequestBody User user) {
        if (!userRepository.exists(user.getEmail())) {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
        User saved = userRepository.save(email, user);
        return new ResponseEntity<>(saved, HttpStatus.CREATED);
    }

    @RequestMapping(value = "/user/{email}", method = RequestMethod.DELETE)
    public ResponseEntity<User> deleteUser(@PathVariable String email)
    {
        if (!userRepository.exists(email)) {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
        userRepository.delete(email);
        return new ResponseEntity<>(HttpStatus.OK);
    }
}
```

You can see that we evolve towards the first level of RESTfulness but there is a lot of boilerplate code involved.

Status codes with exceptions

Another way to handle errors in our API is to throw exceptions. There are two ways to map exceptions with Spring MVC:

- Using `@ExceptionHandler` at the class level, like we did for `IOException` in our upload controller in *Chapter 4, File Upload and Error Handling*
- Using `@ControllerAdvice` to catch global exceptions thrown by all controllers or a subset of your controllers

These two options help you make some business-oriented decisions and define a set of practices within your application.

To associate these handlers with HTTP status codes, we can either inject the response in the annotated method and use the `HttpServletResponse.sendError()` method or just annotate the method with the `@ResponseStatus` annotation.

We will define our own exception, `EntityNotFoundException`. Our business repositories will throw this exception when the entity the user is working on cannot be found. This will help relieve the API code.

Here is the code for the exception. We can put it in a new package called `error`:

```
package masterSpringMvc.error;

public class EntityNotFoundException extends Exception {
    public EntityNotFoundException(String message) {
        super(message);
    }

    public EntityNotFoundException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

Our repository will now throw exceptions in various locations. We will also differentiate between saving and updating a user:

```
package masterSpringMvc.user;

import masterSpringMvc.error.EntityNotFoundException;
import org.springframework.stereotype.Repository;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

@Repository
public class UserRepository {
    private final Map<String, User> userMap = new
    ConcurrentHashMap<>();

    public User update(String email, User user) throws
    EntityNotFoundException {
        if (!exists(email)) {
            throw new EntityNotFoundException("User " + email + " "
cannot be found");
        }
        user.setEmail(email);
        return userMap.put(email, user);
    }

    public User save(User user) {
        return userMap.put(user.getEmail(), user);
    }

    public User findOne(String email) throws EntityNotFoundException {
        if (!exists(email)) {
            throw new EntityNotFoundException("User " + email + " "
cannot be found");
        }
        return userMap.get(email);
    }

    public List<User> findAll() {
        return new ArrayList<>(userMap.values());
    }
}
```

```
public void delete(String email) throws EntityNotFoundException {
    if (!exists(email)) {
        throw new EntityNotFoundException("User " + email + " "
cannot be found");
    }
    userMap.remove(email);
}

public boolean exists(String email) {
    return userMap.containsKey(email);
}
}
```

Our controller becomes simpler since it doesn't have to handle the 404 status. We now throw the `EntityNotFoundException` from our controller methods:

```
package masterSpringMvc.user.api;

import masterSpringMvc.error.EntityNotFoundException;
import masterSpringMvc.user.User;
import masterSpringMvc.user.UserRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api")
public class UserApiController {

    private UserRepository userRepository;

    @Autowired
    public UserApiController(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    @RequestMapping(value = "/users", method = RequestMethod.GET)
    public List<User> findAll() {
        return userRepository.findAll();
    }
}
```

```
    @RequestMapping(value = "/users", method = RequestMethod.POST)
    public ResponseEntity<User> createUser(@RequestBody User user) {
        HttpStatus status = HttpStatus.OK;
        if (!userRepository.exists(user.getEmail())) {
            status = HttpStatus.CREATED;
        }
        User saved = userRepository.save(user);
        return new ResponseEntity<>(saved, status);
    }

    @RequestMapping(value = "/user/{email}", method = RequestMethod.PUT)
    public ResponseEntity<User> updateUser(@PathVariable String email,
    @RequestBody User user) throws EntityNotFoundException {
        User saved = userRepository.update(email, user);
        return new ResponseEntity<>(saved, HttpStatus.CREATED);
    }

    @RequestMapping(value = "/user/{email}", method = RequestMethod.DELETE)
    public ResponseEntity<User> deleteUser(@PathVariable String email)
    throws EntityNotFoundException {
        userRepository.delete(email);
        return new ResponseEntity<>(HttpStatus.OK);
    }
}
```

If we don't handle this exception, Spring will throw a 500 error by default. To handle it we will create a small class in the error package, right next to our EntityNotFoundException class. It will be called `EntityNotFoundMapper` class and will be in charge of handling the exception:

```
package masterSpringMvc.error;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseStatus;

@ControllerAdvice
public class EntityNotFoundMapper {

    @ExceptionHandler(EntityNotFoundException.class)
    @ResponseStatus(value = HttpStatus.NOT_FOUND, reason = "Entity
    could not be found")
```

```
public void handleNotFound() {
}
}
```

The `@ControllerAdvice` annotation allows us to add some behaviors to a set of controllers by annotating a bean. Those controller advice can handle exceptions but also declare model attributes with `@ModelAttribute` or validator policies with `@InitBinder`.

With the code we just wrote, we handle all the `EntityNotFoundException` class thrown by our controllers in one place and associate it with the 404 status. That way, we can abstract this notion and ensure that our application will handle it consistently in all controllers.

We are not going to deal with hyperlinks in our API at our level. Instead, I encourage you to have a look at Spring HATEOAS and Spring Data REST, which provide very elegant solutions to make your resources more discoverable.

Documentation with Swagger

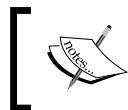
Swagger is a really awesome project that will allow you to document and interact with your API within an HTML5 webpage. The following screenshot illustrates the API documentation:

The screenshot shows the Swagger UI interface. At the top, there's a header with 'Api Documentation' and links for 'Api Documentation', 'Created by Contact Email', and 'Apache 2.0'. Below the header, there are three sections representing different controllers:

- documentation-controller**: Shows a single endpoint: `GET /api/search/{searchType}`. There are 'Show/Hide', 'List Operations', and 'Expand Operations' buttons.
- search-api-controller**: Shows a single endpoint: `GET /api/search/{searchType}`. There are 'Show/Hide', 'List Operations', and 'Expand Operations' buttons.
- user-api-controller**: Shows four endpoints:
 - `PUT /api/user/{email}` (highlighted in orange, labeled 'updateUser')
 - `DELETE /api/user/{email}` (highlighted in red, labeled 'deleteUser')
 - `GET /api/users` (highlighted in blue, labeled 'findAll')
 - `POST /api/users` (highlighted in green, labeled 'createUser')

At the bottom of the interface, there's a note: '[BASE URL: /, API VERSION: 1.0]'.

Swagger used to be big (written in Scala) and somewhat complicated to configure with a Spring setup. Since version 2.0, the library has been rewritten and a really neat project called `spring-fox` will allow for easy integration.



`spring-fox`, formerly known as `swagger-springmvc`, has been in existence for more than three years and is still a very active project.



Add the following dependencies to your build file:

```
compile 'io.springfox:springfox-swagger2:2.1.2'  
compile 'io.springfox:springfox-swagger-ui:2.1.2'
```

The first one will provide an annotation to enable Swagger in your application as well as an API to describe your resources with annotations. Swagger will then generate a JSON representation of your API.

The second is a WebJar that contains static resources consuming the generated JSON through a web client.

The only thing you need to do now is add the `@EnableSwagger2` annotation to your `WebConfiguration` class:

```
@Configuration  
@EnableSwagger2  
public class WebConfiguration extends WebMvcConfigurerAdapter {  
}
```

The `swagger-ui.jar` file we just added contains an HTML file in `META-INF/resources`.

It will automatically be served by Spring Boot when you go to `http://localhost:8080/swagger-ui.html`.

By default, Springfox will scan your whole classpath and show all the request mappings declared in your application.

In our case, we only want to expose the API:

```
@Bean
public Docket userApi() {
    return new Docket(DocumentationType.SWAGGER_2)
        .select()
        .paths(path -> path.startsWith("/api/"))
        .build();
}
```

Springfox works with groups of Dockets that you have to define as beans in your configuration classes. They are logical grouping for RESTful resources. An application can have many of them.

Have a look at the documentation (<http://springfox.github.io/springfox>) to see all the different setups available.

Generating XML

RESTful APIs sometimes return responses in different media types (JSON, XML, and so on). The mechanism responsible for choosing the correct media type is known as content negotiation in Spring.

By default, in Spring MVC, the `ContentNegotiatingViewResolver` bean will be in charge of resolving the correct content according to the content negotiation policies defined in your application.

You can have a look at `ContentNegotiationManagerFactoryBean` to see how these policies are applied within Spring MVC.

Content type can be resolved with the following strategies:

- According to the `Accept` header sent by the client
- With a parameter such as `?format=json`
- With a path extension such as `/myResource.json` or `/myResource.xml`

You can customize these strategies in your Spring configuration by overriding the `configureContentNegotiation()` method of the `WebMvcConfigurerAdapter` class.

By default, Spring will use the `Accept` header and the path extension.

To enable XML serialization with Spring Boot, you can add the following dependency to your classpath:

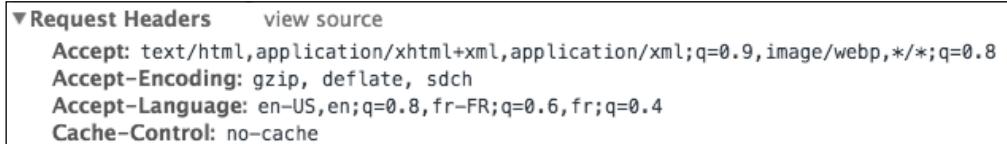
```
compile 'com.fasterxml.jackson.dataformat:jackson-dataformat-xml'
```

If you explore your API with your browser and go to

`http://localhost:8080/api/users`, you will see the result as XML, as follows:



That's because your browser doesn't usually request JSON, but XML is second after HTML. This is shown in the following screenshot:



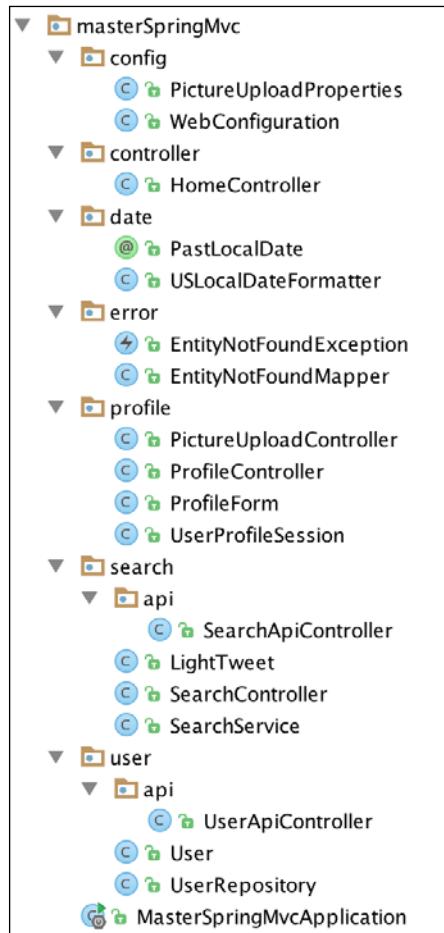
To get JSON back, you can either go to `http://localhost:8080/api/users.json` or send the appropriate `Accept` header with Postman or `httpie`.

The check point

In this chapter, we added a search `ApiController` class. Because the tweets returned by the Twitter API were not adapted to our usage, we introduced a `LightTweet` class to transform them into a friendlier format.

We also developed a user API. The `User` class is the model. The users are stored and retrieved via the `UserRepository` class, and the `UserApiController` class exposes HTTP endpoints to perform CRUD operations on the users. We also added a generic exception and a mapper to associate the exception to an HTTP status.

In the configuration, we added a bean that documents our API, thanks to Swagger, and we customized the serialization of our JSR-310 dates. Our code base should look like the following:



Summary

In this chapter, we have seen how to create a RESTful API with Spring MVC. This kind of backend yields great benefits in terms of performance and maintenance and can do wonders when coupled with a JavaScript MVC framework such as Backbone, Angular JS, or React.js.

We saw how to handle errors and exceptions properly and learned how to leverage the HTTP status to make a better API.

Finally we added automatic documentation with Swagger and added the ability to produce both XML and JSON.

In the next chapter, we will learn how to secure our application as well as use the Twitter API to sign our users up.

6

Securing Your Application

In this chapter, we'll learn how to secure our web application and also how to cope with the security challenges of modern, distributed web applications.

This chapter will be broken up into five parts:

- First, we will set up basic HTTP authentication in a few minutes
- Then, we will design a form-based authentication for the web pages, keeping the basic authentication for the RESTful API
- We will allow the users to sign up via the Twitter OAuth API
- Then, we will leverage Spring Session to make sure our application can scale using a distributed session mechanism
- Finally, we will configure Tomcat to use a secured connection through SSL

Basic authentication

The simplest possible authentication mechanism is basic authentication (http://en.wikipedia.org/wiki/Basic_access_authentication). In a nutshell, our pages will not be available without username and password.

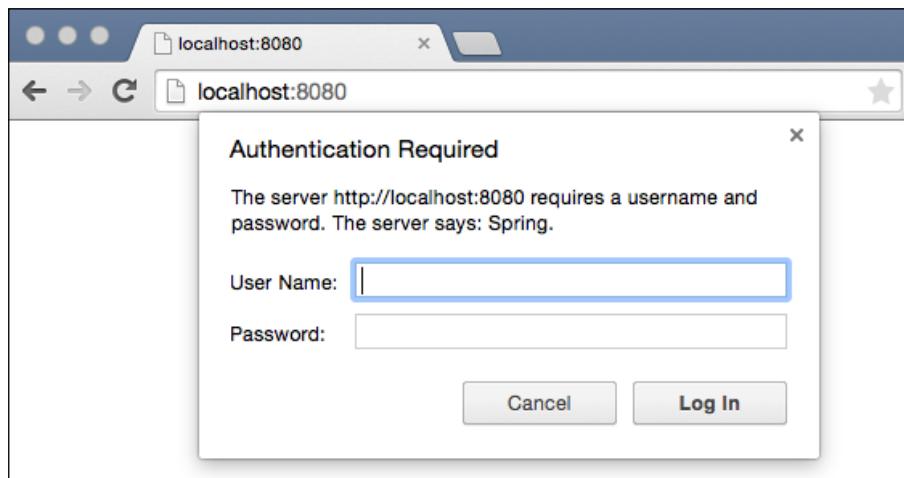
Our server will indicate our resources are secured by sending the `401 Not Authorized` HTTP status code and generate a `WWW-Authenticate` header.

To successfully pass the security check, the client must send an `Authorization` header containing the `Basic` value followed by a base 64 encoding of the `user:password` string. A browser window will prompt the user for a username and a password, granting them access to the secured pages if authentication is successful.

Let's add Spring Security to our dependencies:

```
compile 'org.springframework.boot:spring-boot-starter-security'
```

Relaunch your application and navigate to any URL in your application. You will be prompted for a username and a password:



If you fail to authenticate, you will see that a 401 error is thrown. The default username is user. The correct password for authentication will be randomly generated each time the application launches and will be displayed in the server log:

```
Using default security password: 13212bb6-8583-4080-b790-103408c93115
```

By default, Spring Security secures every resource except a handful of classic routes such as /css/, /js/, /images/, and **/favicon.ico.

If you wish to configure the default credentials, you can add the following properties to the application.properties file:

```
security.user.name=admin  
security.user.password=secret
```

Authorized users

Having only one user in our application does not allow fine-grained security. If we wanted more control over the user credentials, we could add the following `SecurityConfiguration` class in the `config` package:

```
package masterSpringMvc.config;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.method.configuration.EnableGlobalMethodSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

@Configuration
@EnableGlobalMethodSecurity(securedEnabled = true)
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Autowired
    public void configureAuth(AuthenticationManagerBuilder auth)
        throws Exception {
        auth.inMemoryAuthentication()
            .withUser("user").password("user").roles("USER").and()
            .withUser("admin").password("admin").roles("USER",
"ADMIN");
    }
}
```

This snippet will set up an in-memory system containing our application's users as well as their roles. It will override the security name and password previously defined in the application's properties.

The `@EnableGlobalMethodSecurity` annotation will allow us to annotate our application's method and classes to define their security level.

For example, let's say that only the administrators of our application can access the user API. In this case, we just have to add the `@Secured` annotation to our resource to allow access only to ADMIN roles:

```
@RestController
@RequestMapping("/api")
@Secured("ROLE_ADMIN")
public class UserApiController {
    // ... code omitted
}
```

We can easily test that with `httpie` by using the `-a` switch to use basic authentication and the `-p=h` switch, which will only display the response headers.

Let's try this with a user without the admin profile:

```
> http GET 'http://localhost:8080/api/users' -a user:user -p=h
HTTP/1.1 403 Forbidden
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Content-Type: application/json; charset=UTF-8
Date: Sat, 23 May 2015 17:40:09 GMT
Expires: 0
Pragma: no-cache
Server: Apache-Coyote/1.1
Set-Cookie: JSESSIONID=2D4761C092EDE9A4DB91FA1CAA16C59B; Path=/; HttpOnly
Transfer-Encoding: chunked
X-Content-Type-Options: nosniff
X-Frame-Options: DENY
X-XSS-Protection: 1; mode=block
```

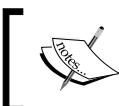
Now, with the administrator:

```
> http GET 'http://localhost:8080/api/users' -a admin:admin -p=h
HTTP/1.1 200 OK
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Content-Type: application/json; charset=UTF-8
Date: Sat, 23 May 2015 17:42:58 GMT
Expires: 0
Pragma: no-cache
Server: Apache-Coyote/1.1
```

```
Set-Cookie: JSESSIONID=CE7A9BF903A25A7A8BAD7D4C30E59360; Path=/; HttpOnly
Transfer-Encoding: chunked
X-Content-Type-Options: nosniff
X-Frame-Options: DENY
X-XSS-Protection: 1; mode=block
```

You will also notice that Spring Security automatically added some common security headers:

- Cache Control: This prevents the user from caching secured resources
- X-XSS-Protection: This tells the browser to block what looks like CSS
- X-Frame-Options: This disallows our site from being embedded in an IFrame
- X-Content-Type-Options: This prevents browsers from guessing the MIME types of malicious resources used to forge XSS attacks



A comprehensive list of these headers is available at <http://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/#headers>.



Authorized URLs

Annotating our controller is very easy but isn't always the most viable option. Sometimes, we just want total control over our authorization.

Remove the `@Secured` annotation; we will come up with something better.

Let's see what Spring Security will allow us to do by modifying the `SecurityConfiguration` class:

```
@Configuration
@EnableGlobalMethodSecurity(securedEnabled = true)
public class SecurityConfiguration extends
WebSecurityConfigurerAdapter {

    @Autowired
    public void configureAuth(AuthenticationManagerBuilder auth)
        throws Exception {
        auth.inMemoryAuthentication()
            .withUser("user").password("user").roles("USER").and()
            .withUser("admin").password("admin").roles("USER",
"ADMIN");
    }
}
```

```
}

@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .httpBasic()
        .and()
        .csrf().disable()
        .authorizeRequests()
        .antMatchers("/login", "/logout").permitAll()
        .antMatchers(HttpMethod.GET, "/api/**").hasRole("USER")
        .antMatchers(HttpMethod.POST, "/api/**").hasRole("ADMIN")
        .antMatchers(HttpMethod.PUT, "/api/**").hasRole("ADMIN")
        .antMatchers(HttpMethod.DELETE, "/api/**").
        hasRole("ADMIN")
        .anyRequest().authenticated();
    }
}
```

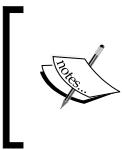
In the preceding code sample, we configured our application's security policy by using Spring Security's fluent API.

This API allows us to configure Spring Security globally by invoking methods associated with different security concerns and chaining with the `and()` method.

What we just defined is a basic authentication, without CSRF protection. Requests on `/login` and `/logout` will be allowed for all users. GET requests on the API will only be permitted for users with the `USER` role, whereas POST, PUT, and DELETE requests on the API will only be accessible to users with the `ADMIN` roles. Finally, every other request will require authentication with any role.

CSRF stands for **Cross Site Request Forgery** and refers to an attack where a malicious website would display a form on its website and post the form data on yours. If the user of your site is not signed out, the POST request would retain the user cookies and would therefore be authorized.

CSRF protection will generate short-lived tokens that will be posted along with the form data. We will see how to properly enable it in the next section; for now, let's just disable it. See <http://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/#csrf> for more details.



To learn more about the authorize request API, have a look at <http://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/#authorize-requests>.

Thymeleaf security tags

Sometimes, you will need to display data coming from the authentication layer, for example the user's name and roles, or hide and display part of a web page according to users' authorities. The `thymeleaf-extras-springsecurity` module will allow us to do so.

Add the following dependency to your `build.gradle` file:

```
compile 'org.thymeleaf.extras:thymeleaf-extras-springsecurity3'
```

With this library, we can add a little block under our navigation bar in `layout/default.html` to display the logged-in user:

```
<!DOCTYPE html>
<html xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      xmlns:th="http://www.thymeleaf.org"
      xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-
springsecurity3">
<head>
    <!-- content trimmed -->
</head>
<body>

    <!-- content trimmed -->
<nav>
    <div class="nav-wrapper indigo">
        <ul class="right">
            <!-- content trimmed -->
        </ul>
    </div>
</nav>
<div>
    You are logged as <b sec:authentication="name" /> with roles <span
    sec:authentication="authorities" />
    -
    <form th:action="@{/logout}" method="post" style="display: inline-
block">

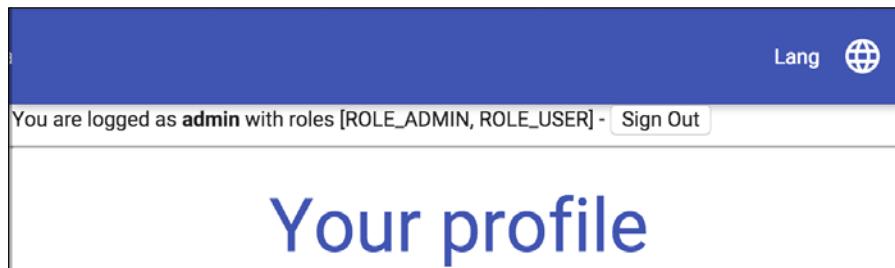
```

```
<input type="submit" value="Sign Out" />
</form>
<hr/>
</div>

<section layout:fragment="content">
    <p>Page content goes here</p>
</section>

<!-- content trimmed -->
</body>
</html>
```

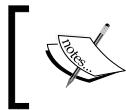
Note the new namespace in the HTML declaration and the `sec:authentication` attributes. It allows access to the properties of the `org.springframework.security.core.Authentication` object, which represents the user who is currently logged in, as shown in the following screenshot:



Don't click on the logout link just yet as it doesn't work with basic authentication. We will get it to work in the next part.

The `lib` tag also has a handful of other tags, such as the one to check user authorizations:

```
<div sec:authorize="hasRole('ROLE_ADMIN')">
    You are an administrator
</div>
```



Please refer to the documentation available at <https://github.com/thymeleaf/thymeleaf-extras-springsecurity> to learn more about the library.

The login form

Basic authentication is good for our RESTful API, but we would rather have a login page carefully designed by our team to improve the web experience.

Spring Security allows us to define as many `WebSecurityConfigurerAdapter` classes as we need. We will split our `SecurityConfiguration` class into two parts:

- `ApiSecurityConfiguration`: This will be configured first. This will secure the RESTful endpoints with basic authentication.
- `WebSecurityConfiguration`: This will then configure login form for the rest of our application.

You can remove or rename `SecurityConfiguration` and create `ApiSecurityConfiguration` instead:

```

@Configuration
@Order(1)
public class ApiSecurityConfiguration extends
WebSecurityConfigurerAdapter {

    @Autowired
    public void configureAuth(AuthenticationManagerBuilder auth)
        throws Exception {
        auth.inMemoryAuthentication()
            .withUser("user").password("user").roles("USER").and()
            .withUser("admin").password("admin").roles("USER",
"ADMIN");
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .antMatcher("/api/**")
            .httpBasic().and()
            .csrf().disable()
            .authorizeRequests()
            .antMatchers(HttpMethod.GET).hasRole("USER")
            .antMatchers(HttpMethod.POST).hasRole("ADMIN")
            .antMatchers(HttpMethod.PUT).hasRole("ADMIN")
            .antMatchers(HttpMethod.DELETE).hasRole("ADMIN")
            .anyRequest().authenticated();
    }
}

```

Note the `@Order(1)` annotation, which will ensure that this configuration is executed before the other one. Then, create a second configuration for the web, called `WebSecurityConfiguration`:

```
package masterSpringMvc.config;

import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

@Configuration
public class WebSecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .formLogin()
            .defaultSuccessUrl("/profile")
            .and()
            .logout().logoutSuccessUrl("/login")
            .and()
            .authorizeRequests()
            .antMatchers("/webjars/**", "/login").permitAll()
            .anyRequest().authenticated();
    }
}
```

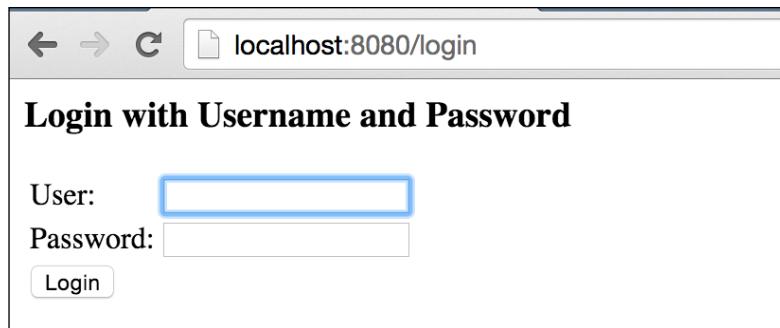
The result of this code is that anything matching `/api/**` will be secured with basic authentication, without CSRF protection. Then, the second configuration will be loaded. It will secure anything else. Everything in this part of the application requires the client to be authenticated, except requests on WebJars and on the login page (this will avoid the redirection loop).

If an unauthenticated user tries to access a protected resource, they will automatically be redirected to the login page.

By default, the login URL is `GET /login`. The default login will be posted via a `POST /login` request that will contain three values: a user name (`username`), a password (`password`) and a CSRF token (`_csrf`). If the login is unsuccessful, the user will be redirected to `/login?error`. The default logout page is a `POST /logout` request with a CSRF token.

Now, if you try to navigate on your application, this form will be generated automatically!

If you are already logged in from a previous attempt, close your browser; this will clear up the session.



We can now log in and out of our application!

This is lovely but we can do a lot better with very little effort. First, we will define a login page on `/login` in the `WebSecurityConfiguration` class:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .formLogin()
        .loginPage("/login") // <= custom login page
        .defaultSuccessUrl("/profile")
        // the rest of the configuration stays the same
}
```

This will let us create our own login page. To do that, we will need a very simple controller to handle the `GET` `login` request. You can create one in the `authentication` package:

```
package masterSpringMvc.authentication;

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

@Controller
public class LoginController {

    @RequestMapping("/login")
    public String authenticate() {
```

```
        return "login";
    }
}
```

This will trigger the display of the `login.html` page located in the template directory. Let's create it:

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      layout:decorator="layout/default">
<head>
    <title>Login</title>
</head>
<body>
<div class="section no-pad-bot" layout:fragment="content">
    <div class="container">

        <h2 class="header center orange-text">Login</h2>

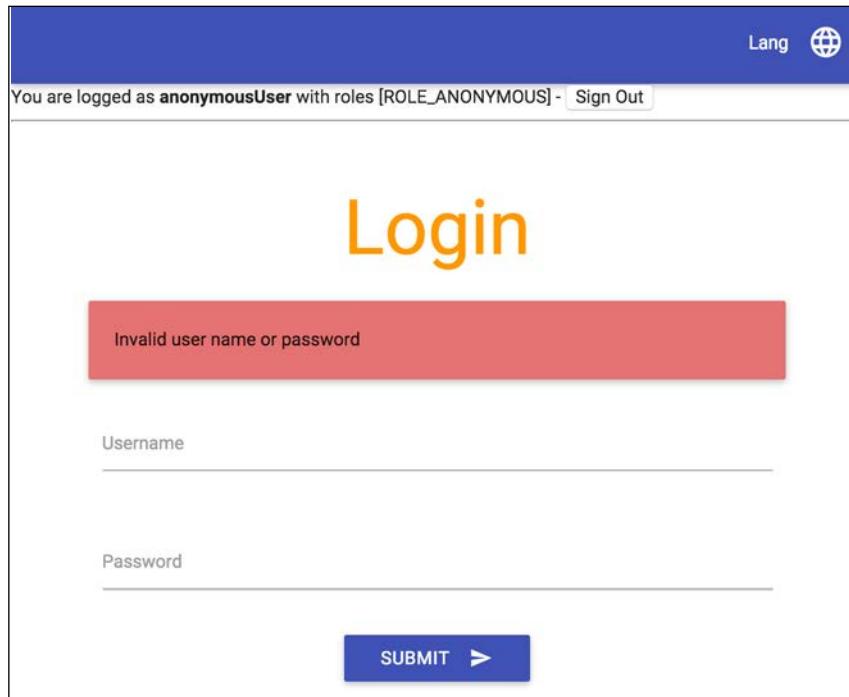
        <div class="row">
            <div id="errorMessage" class="card-panel red lighten-2" th:if="${param.error}">
                <span class="card-title">Invalid user name or
                password</span>
            </div>

            <form class="col s12" action="/login" method="post">
                <div class="row">
                    <div class="input-field col s12">
                        <input id="username" name="username" type="text" class="validate"/>
                        <label for="username">Username</label>
                    </div>
                </div>
                <div class="row">
                    <div class="input-field col s12">
                        <input id="password" name="password" type="password" class="validate"/>
                        <label for="password">Password</label>
                    </div>
                </div>
                <div class="row center">
                    <button class="btn waves-effect waves-light" type="submit" name="action">Submit
                        <i class="mdi-content-send right"></i>
                    </button>
                </div>
                <input type="hidden" th:name="${_csrf.parameterName}"
                      th:value="${_csrf.token}"/>
            </form>
        </div>
    </div>
</body>

```

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

Note that we handle the error message and that we post a CSRF token. We also use the default username and password input names, but those are configurable if needed. The result looks much better already!



You can see right away that Spring Security assigns anonymous credentials to all non-authenticated users by default.

We shouldn't show the sign-out button to an anonymous user so we can wrap the corresponding HTML part in `sec:authorize="isAuthenticated()"` to display it to authenticated users only, like so:

```
<div sec:authorize="isAuthenticated()">
    You are logged as <b sec:authentication="name"/> with roles <span
    sec:authentication="authorities"/>
    -

```

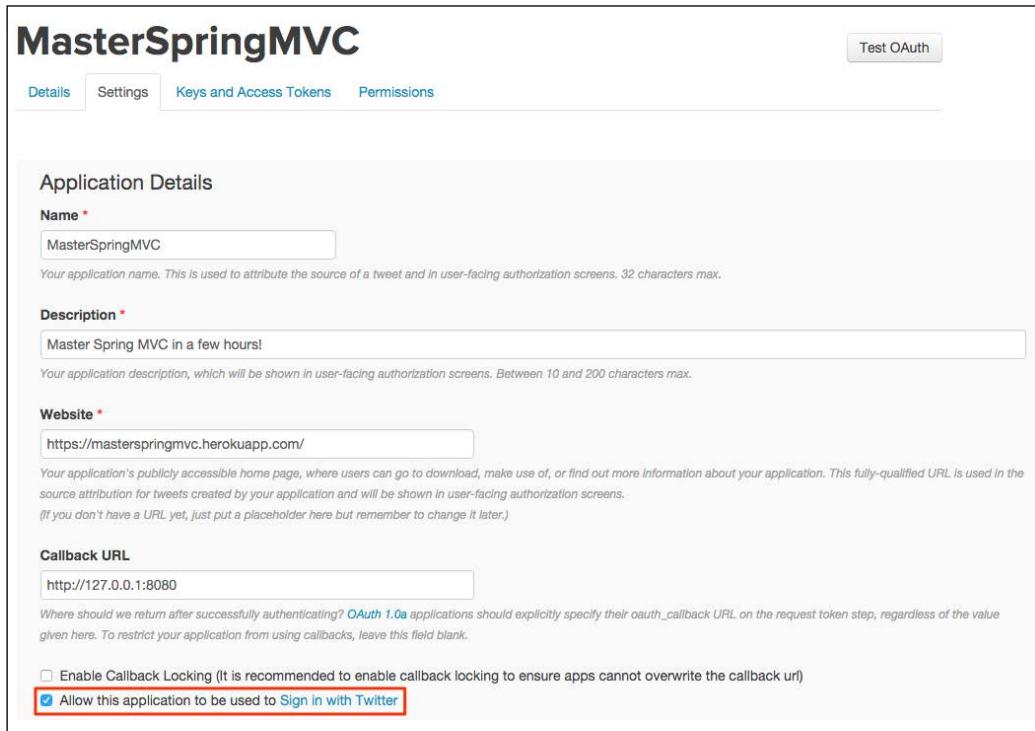
Securing Your Application

```
<form th:action="@{/logout}" method="post" style="display: inline-block">
    <input type="submit" value="Sign Out"/>
</form>
<hr/>
</div>
```

Twitter authentication

Our application is strongly integrated with Twitter, so it seems logical that we would allow authentication through Twitter.

Before going further, make sure that you have enabled Twitter sign in on your app on Twitter (<https://apps.twitter.com>):



The screenshot shows the 'MasterSpringMVC' application settings page on the Twitter developer site. The 'Details' tab is selected. The application details are as follows:

- Name ***: MasterSpringMVC
- Description ***: Master Spring MVC in a few hours!
- Website ***: <https://masterspringmvc.herokuapp.com/>
- Callback URL**: <http://127.0.0.1:8080>

At the bottom of the form, there are two checkboxes:

- Enable Callback Locking (It is recommended to enable callback locking to ensure apps cannot overwrite the callback url)
- Allow this application to be used to Sign in with Twitter

A 'Test OAuth' button is located in the top right corner of the form.

Setting up social authentication

Spring social enables authentication through an OAuth provider such as Twitter through a signin/signup scenario. It will intercept a POST request on /signin/twitter. If the user is not known to the `UsersConnectionRepository` interface, the signup endpoint will be called. It will allow us to take the necessary measures to register the user on our system and maybe ask them for additional details.

Let's get to work. The first thing we need to do is to add the `signin/**` and `/signup` URLs as publicly available resources. Let's modify our `WebSecurityConfiguration` class, changing the `permitAll` line:

```
.antMatchers("/webjars/**", "/login", "/signin/**", "/signup") .
    permitAll()
```

To enable the `signin/signup` scenario, we also need a `SignInAdapter` interface, a simple listener that will be called when an already known user signs in again.

We can create an `AuthenticatingSignInAdapter` class right next to our `LoginController`:

```
package masterSpringMvc.authentication;

import org.springframework.security.authentication.
    UsernamePasswordAuthenticationToken;
import org.springframework.security.core.context.
    SecurityContextHolder;
import org.springframework.social.connect.Connection;
import org.springframework.social.connect.UserProfile;
import org.springframework.social.connect.web.SignInAdapter;
import org.springframework.stereotype.Component;
import org.springframework.web.context.request.NativeWebRequest;

@Component
public class AuthenticatingSignInAdapter implements SignInAdapter {

    public static void authenticate(Connection<?> connection) {
        UserProfile userProfile = connection.fetchUserProfile();
        String username = userProfile.getUsername();
        UsernamePasswordAuthenticationToken authentication = new UsernamePasswordAuthenticationToken(username, null, null);
        SecurityContextHolder.getContext().setAuthentication(authentication);
        System.out.println(String.format("User %s %s connected.", userProfile.getFirstName(), userProfile.getLastName()));
    }
}
```

```
    }

    @Override
    public String signIn(String userId, Connection<?> connection,
NativeWebRequest request) {
    authenticate(connection);
    return null;
}
}
```

As you can see, this handler is called at the perfect time to allow user authentication with Spring Security. We'll come back to that in just a moment. For now, we need to define our `SignupController` class in the same package, the one in charge of first-time visiting users:

```
package masterSpringMvc.authentication;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.social.connect.Connection;
import org.springframework.social.connect.ConnectionFactoryLocator;
import org.springframework.social.connect.UsersConnectionRepository;
import org.springframework.social.connect.web.ProviderSignInUtils;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.context.request.WebRequest;

@Controller
public class SignupController {
    private final ProviderSignInUtils signInUtils;

    @Autowired
    public SignupController(ConnectionFactoryLocat
or connectionFactoryLocator, UsersConnectionRepository
connectionRepository) {
        signInUtils = new ProviderSignInUtils(connectionFactoryLocat
or, connectionRepository);
    }

    @RequestMapping(value = "/signup")
    public String signup(WebRequest request) {
        Connection<?> connection = signInUtils.getConnectionFromSessi
on(request);
        if (connection != null) {
            AuthenticatingSignInAdapter.authenticate(connection);
        }
    }
}
```

```

        signInUtils.doPostSignUp(connection.getDisplayName() ,
request);
    }
    return "redirect:/profile";
}
}

```

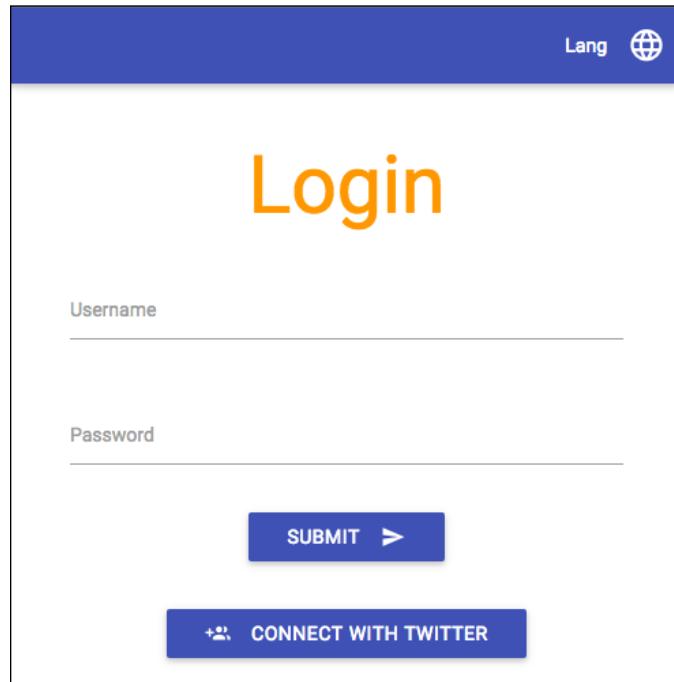
First, this controller retrieves the current connection from the session. Then, it authenticates the user through the same method as before. Lastly, it will trigger the `doPostSignUp` event, which will allow Spring Social to store information relative to our user in the `UsersConnectionRepository` interface that we mentioned earlier.

The last thing we need to do is add a triumphant "login with twitter" button to our login page, right below the previous form:

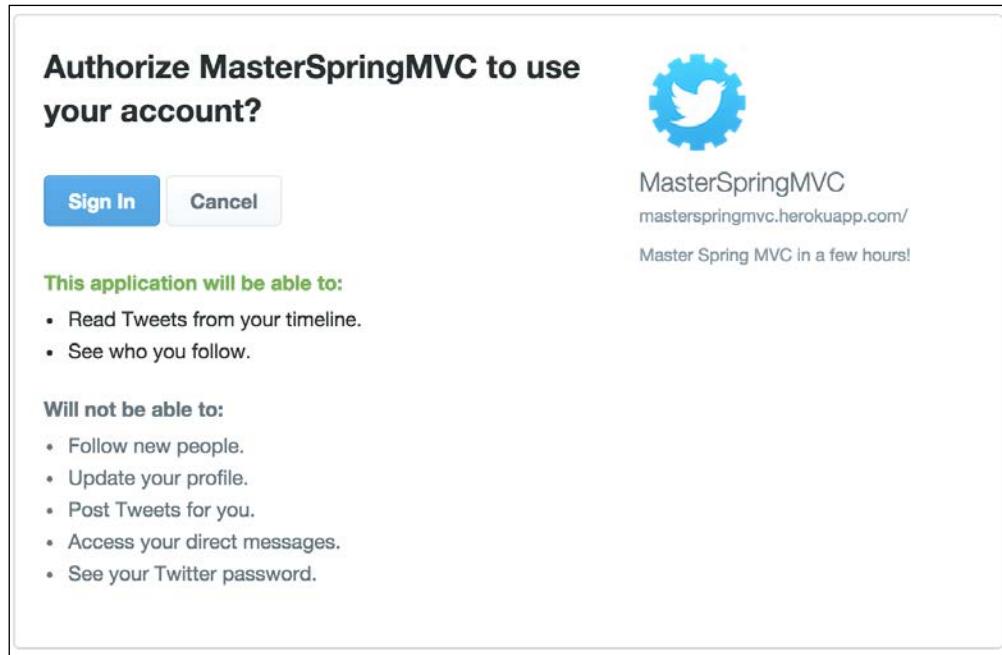
```

<form th:action="@{/signin/twitter}" method="POST" class="center">
    <div class="row">
        <button class="btn indigo" name="twitterSignin"
type="submit">Connect with Twitter
            <i class="mdi-social-group-add left"></i>
        </button>
    </div>
</form>

```



When the user clicks on the **CONNECT WITH TWITTER** button, they will be redirected to a Twitter sign in page:



Explanation

There isn't much code, but it is a bit tricky to understand all the parts. The first step to getting what's going on is to have a look at the `SocialWebAutoConfiguration` class of Spring Boot.

The `SocialAutoConfigurationAdapter` class declared in this class contains the following bean:

```
@Bean
@ConditionalOnBean(SignInAdapter.class)
@ConditionalOnMissingBean(ProviderSignInController.class)
public ProviderSignInController signInController(
    ConnectionFactoryLocator factoryLocator,
    UsersConnectionRepository usersRepository, SignInAdapter
signInAdapter) {
    ProviderSignInController controller = new
    ProviderSignInController(
        factoryLocator, usersRepository, signInAdapter);
    if (!CollectionUtils.isEmpty(this.signInInterceptors)) {
```

```

        controller.setSignInInterceptors(this.signInInterceptors);
    }
    return controller;
}

```

The `ProviderSignInController` class will automatically be set up if one `ProviderSignInController` class is detected in our configuration. This controller is the cornerstone of the sign-in process. Have a look at what it does (I will only summarize the important parts):

- It will handle the `POST /signin/{providerId}` from our connect button
- It will redirect the user to the appropriate sign-in URL of our identification provider
- It will be notified of the OAuth token by a `GET /signin/{providerId}` from the identification provider
- It will then handle the sign in
- If the user is not found in the `UsersConnectionRepository` interface, it will use a `SessionStrategy` interface to store the pending login request and will then redirect to the `signupUrl` page
- If the user is found, the `SignInAdapter` interface is called and the user is redirected to the `postSignupUrl` page

The two important components of this identification are the `UsersConnectionRepository` interface in charge of storing and retrieving users from some kind of storage and the `SessionStrategy` interface that will temporarily store the user connection so it can be retrieved from the `SignupController` class.

By default, Spring Boot creates an `InMemoryUsersConnectionRepository` interface for each authentication provider, which means that our user connection data will be stored in memory. If we restart the server, the user will become unknown and will go through the sign-up process again.

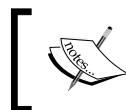
The `ProviderSignInController` class defaults to `HttpSessionSessionStrategy`, which will store the connection in the HTTP session. The `ProviderSignInUtils` class that we use in our `SignupController` class also uses this strategy by default. If we were distributing our application on multiple servers, this would be problematic because the session would likely not be available on every server.

It is easy enough to override these defaults by providing a custom `SessionStrategy` interface to both the `ProviderSignInController` and `ProviderSignInUtils` classes to store data somewhere other than in the HTTP session.

Likewise, we can use another kind of storage for our user connection data by providing another implementation of the `UsersConnectionRepository` interface.

Spring Social provides a `JdbcUsersConnectionRepository` interface that will automatically save authenticated users in a `UserConnection` table in your database. This won't be covered in this book extensively, but you should be able to configure it easily by adding the following bean to your configuration:

```
@Bean
@Primary
public UsersConnectionRepository getUsersConnectionRepository(
    DataSource dataSource, ConnectionFactoryLocator
    connectionFactoryLocator) {
    return new JdbcUsersConnectionRepository(
        dataSource, connectionFactoryLocator, Encryptors.noOpText());
}
```



Check out this article <http://geowarin.github.io/spring/2015/08/02/social-login-with-spring.html> on my blog for more details.



Distributed sessions

As we have seen in the preceding section, there are several moments when Spring Social stores things in the HTTP session. Our user profile is also stored in the session. This is a classical approach to keeping things in memory as long as a user is navigating the site.

However, this can prove troublesome if we want to scale our application and distribute the load to multiple backend servers. We have now entered the cloud era, and *Chapter 8, Optimizing Your Requests* will be about deploying our application to the cloud.

To make our session work in a distributed environment, we have a few options:

- We could use sticky sessions. This will ensure that a specific user will always be redirected to the same server and keep its session. It requires additional configuration for the deployment and isn't a particularly elegant approach.
- Refactor our code to put data in a database instead of the session. We can then load the user's data from the database if we associate it with a cookie or a token sent by the client with each request.
- Use the Spring Session project to transparently use a distributed database such as Redis as the underlying session provider.

In this chapter, we will see how to set up the third approach. It is really easy to set up and provides the amazing benefit that it can be turned off without impacting the functionality of our application.

The first thing we need to do is to install Redis. To install it on Mac, use the brew command:

```
brew install redis
```

For other platforms, follow the instructions at <http://redis.io/download>.

You can then start the server by using the following command:

```
redis-server
```

Add the following dependencies to your build.gradle file:

```
compile 'org.springframework.boot:spring-boot-starter-redis'  
compile 'org.springframework.session:spring-session:1.0.1.RELEASE'
```

Create a new configuration file next to application.properties called application-redis.properties:

```
spring.redis.host=localhost  
spring.redis.port=6379
```

Spring Boot provides a convenient way of associating configuration files with a profile. In this case, the application-redis.properties file will only be loaded if the Redis profile is active.

Then, create a RedisConfig class in the config package:

```
package masterSpringMvc.config;  
  
import org.springframework.context.annotation.Configuration;  
import org.springframework.context.annotation.Profile;  
import org.springframework.session.data.redis.config.annotation.web.  
http.EnableRedisHttpSession;  
  
@Configuration  
@Profile("redis")  
@EnableRedisHttpSession  
public class RedisConfig {  
}
```

As you can see, this configuration will only be active if the redis profile is on.

We're done! We can now launch our app with the following flag:

```
-Dspring.profiles.active=redis
```

You can also generate the JAR with `gradlew build` and launch it with the following command:

```
java -Dserver.port=$PORT -Dspring.profiles.active=redis -jar app.jar
```

Alternatively, you can launch it with Gradle in Bash, as follows:

```
SPRING_PROFILES_ACTIVE=redis ./gradlew bootRun
```

You can also simply set it up as a JVM option in the run configuration of your IDE.

And that's it! You now have a server storing the details of your logged-in users. This means that we can scale and have multiple servers for our web resources and our users won't notice. And we didn't have to write any code on our side.

This also means that you will keep your session even if you restart your server.

To see that it works, connect to Redis with the `redis-cli` command. At the beginning, it will not contain any keys:

```
> redis-cli
127.0.0.1:6379> KEYS *
(empty list or set)
```

Navigate to your app and start putting things in the session:

```
127.0.0.1:6379> KEYS *
1) "spring:session:expirations:1432487760000"
2) "spring:session:sessions:1768a55b-081a-4673-8535-7449e5729af5"
127.0.0.1:6379> HKEYS spring:session:sessions:1768a55b-081a-4673-8535-
7449e5729af5
1) "sessionAttr:SPRING_SECURITY_CONTEXT"
2) "sessionAttr:org.springframework.security.web.csrf.
 HttpSessionCsrfTokenRepository.CSRF_TOKEN"
3) "lastAccessedTime"
4) "maxInactiveInterval"
5) "creationTime"
```



You can consult the list of available commands at
<http://redis.io/commands>.

SSL

Secure Sockets Layer (SSL) is a security protocol in which data is encrypted and sent to a trusted party via a certificate. In this part, I will show you the different ways to create a secured connection with Spring Boot. The completion of these steps is not mandatory to start the next chapter. They are included for completeness, so feel free to skip them if you are in a hurry to deploy your application to the cloud.

In *Chapter 9, Deploying Your Web Application to the Cloud*, we will see that most cloud platforms already handle SSL so we don't have to configure it at our end.

Generating a self-signed certificate

Normally, X.509 certificates are delivered by a Certificate Authority. They generally bill you for the service, so for testing purposes, we can create our own self-signed keystore file.

The JDK comes with a binary called keytool, which is used to manage certificates. With it, you can create a keystore and import certificates into an existing keystore. You can issue the following command inside your project root to create one:

```
$ keytool -genkey -alias masterspringmvc -keyalg RSA -keystore src/main/
resources/tomcat.keystore
Enter keystore password: password
Re-enter new password: password
What is your first and last name?
[Unknown]: Master Spring MVC
What is the name of your organizational unit?
[Unknown]: Packt
What is the name of your organization?
[Unknown]: Packt
What is the name of your City or Locality?
[Unknown]: Paris
What is the name of your State or Province?
[Unknown]: France
What is the two-letter country code for this unit?
[Unknown]: FR
Is CN=Master Spring MVC, OU=Packt, O=Packt, L=Paris, ST=France, C=FR
correct?
[no]: yes
```

```
Enter key password for <masterspringmvc>
(RETURN if same as keystore password): password2
Re-enter new password: password2
```

This will generate a keystore named `masterspringmvc` with the RSA algorithm and will store it in a keystore in `src/main/resources`.



Do not push the keystore to your repository. It can be brute-forced, which would void the security of your website. You should also generate keystores with strong, randomly generated passwords.



The easy way

If all you care about is having one secure https channel and no http channel, it is as easy as it gets:

```
server.port = 8443
server.ssl.key-store = classpath:tomcat.keystore
server.ssl.key-store-password = password
server.ssl.key-password = password2
```



Do not push your passwords to your repository. Use the `{} ${}` notation to import environment variables.



The dual way

If you want to have both the http and the https channels available in your application, you should add this kind of configuration to your application:

```
@Configuration
public class SslConfig {

    @Bean
    public EmbeddedServletContainerFactory servletContainer() throws
    IOException {
        TomcatEmbeddedServletContainerFactory tomcat = new
        TomcatEmbeddedServletContainerFactory();
    }
}
```

```
tomcat.addAdditionalTomcatConnectors(createSslConnector());
    return tomcat;
}

private Connector createSslConnector() throws IOException {
    Connector connector = new Connector(Http11NioProtocol.class.
    getName());
    Http11NioProtocol protocol =
        (Http11NioProtocol) connector.getProtocolHandler();
    connector.setPort(8443);
    connector.setSecure(true);
    connector.setScheme("https");
    protocol.setSSLEnabled(true);
    protocol.setKeyAlias("masterspringmvc");
    protocol.setKeystorePass("password");
    protocol.setKeyPass("password2");
    protocol.setKeystoreFile(new ClassPathResource("tomcat.
    keystore").getFile().getAbsolutePath());
    protocol.setSslProtocol("TLS");
    return connector;
}
}
```

This will load the previously generated keystore to create an additional channel on port 8443 in addition to port 8080.

You can use Spring Security to automatically redirect connections from `http` to `https` with the following configuration:

```
@Configuration
public class WebSecurityConfiguration extends
WebSecurityConfigurerAdapter {

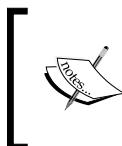
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .requiresChannel().anyRequest().requiresSecure()
            .and()
            /* rest of the configuration */;
    }
}
```

Behind a secured server

The most convenient way to secure your application with SSL is often to put it behind an SSL-enabled web server such as Apache or CloudFlare. These will often use de facto headers to indicate that the connection was previously initiated with SSL.

Spring Boot can understand this protocol if you tell it what the correct headers are in your `application.properties` file:

```
server.tomcat.remote_ip_header=x-forwarded-for  
server.tomcat.protocol_header=x-forwarded-proto
```



See the documentation here for more details at <http://docs.spring.io/spring-boot/docs/current/reference/html/howto-embedded-servlet-containers.html#howto-use-tomcat-behind-a-proxy-server>.

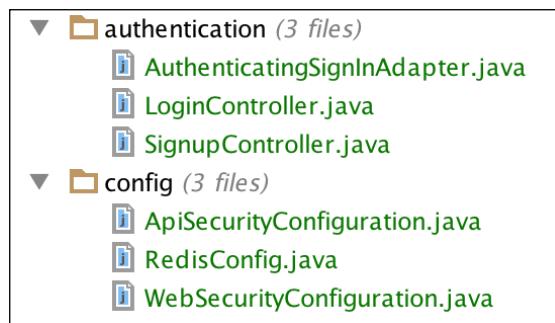


The check point

In this chapter, we added three pieces of configuration:

`ApiSecurityConfiguration`, which configures our REST API to use basic HTTP authentication; `WebSecurityConfiguration`, which sets up a login form for our web users to sign in with either an account or with Twitter; and `RedisConfig`, which allows our sessions to be stored and retrieved from a Redis server.

In the authentication package, we added a `LoginController` class that redirects to our login page, a `SignupController` class that will be called the first time a user signs up with Twitter, and an `AuthenticatingSignInAdapter` class that will be called on every login with Twitter:



Summary

Securing our web application with Spring is really simple. The possibilities are endless, and advanced configurations such as social sign in are at your fingertips. Distributing sessions and scaling also take a matter of minutes.

In the next chapter, we will see how to test our application and ensure it never regresses.

7

Leaving Nothing to Luck – Unit Tests and Acceptance Tests

In this chapter, we will see why and how our application should be tested. We will see the differences between unit tests and acceptance tests, and learn how to do both.

This chapter is divided in two parts. In the first part, we will write tests in Java while studying the different ways of testing. In the second part, which is shorter, we will write the exact same tests in Groovy, and see how we can improve our code readability with this awesome language.

If you do everything in this chapter, you will have double tests, so feel free to keep only the tests that are most readable for you.

Why should I test my code?

Working in the Java world has made a lot of developers aware of the importance of tests. A good series of tests can catch regressions early and allows us to be more confident when we ship our product.

A lot of people are now familiar with the notion of continuous integration (<http://www.thoughtworks.com/continuous-integration>). This is a practice where a server is in charge of building the application every time a change is made on the source control system.

The build should be as fast as possible and capable of self testing. The main idea of this practice is to get a fast feedback loop; you should get details about what went wrong as soon as something in the system breaks.

Why should you care? After all, testing your application is an additional cost; the time spent designing and maintaining tests will necessarily eat into some development time.

Actually, the later a bug is found, the costlier it gets. If you think about it, even a bug found by your QA team begins to cost more than a bug you find on your own. It forces you to switch back to the context you were in when writing the code: why did I write this line? What was the underlying business rule of that function?

If you write tests early on and are able to launch them in a few seconds, it will certainly cost less time to address potential bugs in your code.

Another benefit of tests is that they act as a living documentation of your code. While writing extensive documentation, and even code comments, can prove ineffective because they easily become outdated, forming the habit of writing a good test for limit cases or surprising behaviors will act as a safety net for the future.

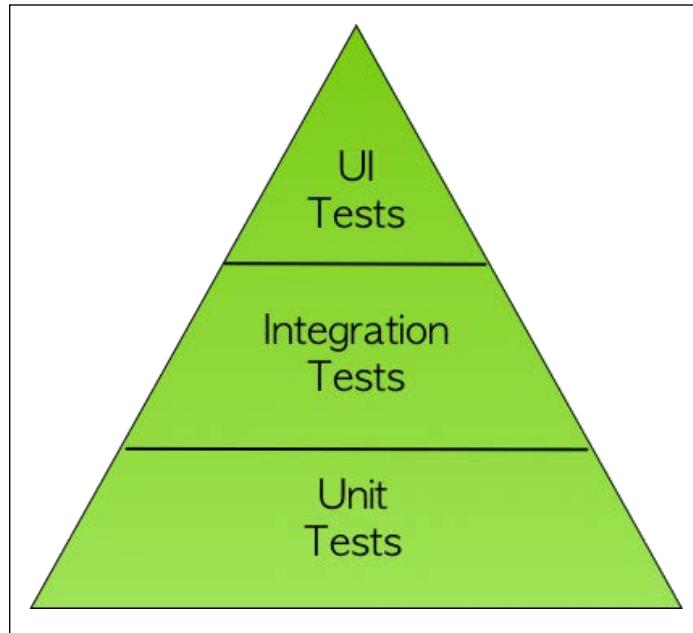
What is this line of code for? Have you ever found yourself asking this kind of question? Well, if you have a good set of unit tests, you can just remove it and see what breaks! Tests give us an unprecedented confidence in our code and in our ability to refactor it. Software is very fragile. If you stop caring, it will slowly rot and die.

Be responsible – don't let your code die!

How should I test my code?

There are different kinds of tests that we can perform on a piece of software, such as security tests, performances test, and so on. As developers, we will focus on the tests we can automate and that will help improve our code.

The tests fall under two categories: unit tests and acceptance tests. The test pyramid (<http://martinfowler.com/bliki/TestPyramid.html>) shows in what proportions these tests should be written:

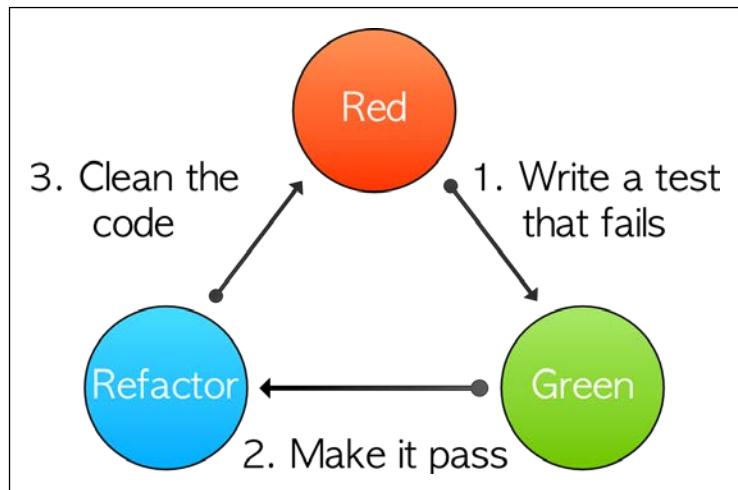


At the bottom of the pyramid, you have the unit tests (fast to launch and relatively easy to maintain), and at the top, UI tests (costlier and slower to execute). Integration tests sit in the middle: they can be viewed as big unit tests with complex interactions between units.

The idea of the pyramid is to remind you to put your focus where you have the most impact and get the best feedback loops.

Test-driven development

Many developers develop the healthy habit of **Test-driven Development (TDD)**. This practice, inherited from Extreme Programming (XP), is the process of splitting each development stage into small steps and then writing a failing test for every one of them. You make the necessary modifications so that the tests pass again (tests are green). You can then refactor your code as long as the tests remain green. The following figure illustrates the TDD cycle:



You can iterate until the feature is done with very short feedback loops, with the insurance of no regressions, and the guarantee that all the code you write will be tested from the very beginning.

TDD gets its share of criticisms. The most interesting ones are these:

- It takes more time to write the tests than to do the actual implementation
- It can lead to poorly designed applications

The truth of the matter is that it takes time to become a good TDD practitioner. Once you get the feeling of what should be tested and know your tools well enough, you won't lose much time at all.

It also takes experienced developers to craft an application with a proper design using TDD (or with any other methodology). Poor design can be a side effect of TDD if you get trapped in the baby steps mantra and forget to look at the big picture. It is true that TDD won't magically lead to great application design, so be careful and remember to take a step back after completing each feature.

From the beginning of the book, we have only had one autogenerated unit test in our code. This is bad! We didn't follow good practice. This chapter is here to address this problem.

The unit tests

The lower level tests we can write are called unit tests. They should test a small portion of code, hence the notion of unit. How you define a unit is up to you; it can be a class or a bunch of closely related classes. Defining this notion will determine what will be mocked (replaced with a dummy object). Are you going to replace the database with a lightweight alternative? Are you going to replace interactions with external services? Are you going to mock-up closely related objects whose behavior is not relevant to the context of what's being tested?

My advice here is to keep a balanced approach. Keep your tests clean and fast, and everything else will follow.

I rarely completely mock the data layer. I tend to use embedded databases for testing. They provide an easy way to load data while testing.

As a rule, I always mock collaboration with external services for two reasons, as follows:

- The speed of the tests and the possibility to run the tests without connecting to the network
- To be able to test error cases while communicating with those services

Additionally, there is a subtle difference between mocking and stubbing. We will try to use both approaches to see how they relate to each other.

The right tools for the job

The first barrier for test novices is the lack of knowledge of the good tools and libraries for writing relevant and maintainable tests.

I'm going to list a few here. This list is by no means exhaustive, but it contains the tools we are going to use and that are easily compatible with Spring:

JUnit	The most universally adopted Java test runner. Launched by default by all build tools.
AssertJ	A fluent assertion library. It's way easier to use than Hamcrest.
Mockito	An easy mocking framework.
DbUnit	For mocking and asserting your database content with XML datasets.
Spock	An elegant Groovy DSL to write tests with Behaviour Driven Development (BDD) style (Given/When/Then).

Groovy has a place of choice in my testing toolset. Even if you're not ready yet to put some Groovy code into production, you can easily use the convenience of the language in your tests. With Gradle, this is very easy to do, but we will see that in a few minutes.

The acceptance tests

In the context of a web application, "acceptance test" will often refer to in-browser, end-to-end testing. In the Java world, Selenium is clearly one of the most reliable and mature libraries.

In the JavaScript world, we can find other alternatives, such as PhantomJS or Protractor. PhantomJS is very relevant in our case because there is a web driver available to run Selenium tests inside of this headless browser, which will improve launch time and won't require emulating an X Server or launching a separate Selenium server:

Selenium 2	This provides web drivers to pilot browsers for automated testing.
PhantomJS	A headless browser (without GUI). Probably the fastest browser.
FluentLenium	A fluent library for piloting Selenium tests.
Geb	A Groovy library for piloting Selenium tests.

Our first unit test

It is now time to write our first unit test.

We will focus on writing tests at the controller level because we have little to no business code or service. The key to writing tests for Spring MVC is the `org.springframework.boot:spring-boot-starter-test` dependency in our classpath. It will add a few very useful libraries, such as these:

- `hamcrest`: This is JUnit's assertion library
- `mockito`: This is a mocking library
- `spring-test`: This is the Spring testing library

We will test the redirection to the profile page that is created when the user hasn't created their profile yet.

We already have an autogenerated test called `MasterSpringMvc4ApplicationTests`. It is the most basic kind of test one can write with the Spring test framework: it does nothing but blow up if the context cannot be loaded:

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = MasterSpringMvc4Application.class)
@WebAppConfiguration
public class MasterSpringMvc4ApplicationTests {

    @Test
    public void contextLoads() {
    }
}
```

We can delete this test and create one that will ensure that a user with no profile will be redirected to the profile page by default. It will actually test the code of the `HomeController` class, so let's call it `HomeControllerTest` class and put it in the same package as `HomeController`, in `src/test/java`. All IDEs have shortcuts for creating a JUnit test case from a class. Find out how to do it with yours now!

Here is the test:

```
package masterSpringMvc.controller;

import masterSpringMvc.MasterSpringMvcApplication;
import org.junit.Before;
import org.junit.Test;
```

```
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.SpringApplicationConfiguration;
import org.springframework.test.context.junit4.
SpringJUnit4ClassRunner;
import org.springframework.test.context.web.WebAppConfiguration;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;
import org.springframework.web.context.WebApplicationContext;

import static org.springframework.test.web.servlet.request.
MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.
MockMvcResultHandlers.print;
import static org.springframework.test.web.servlet.result.
MockMvcResultMatchers.*;

@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = MasterSpringMvcApplication.
class)
@WebAppConfiguration
public class HomeControllerTest {
    @Autowired
    private WebApplicationContext wac;

    private MockMvc mockMvc;

    @Before
    public void setup() {
        this.mockMvc = MockMvcBuilders.webAppContextSetup(this.wac) .
build();
    }

    @Test
    public void should_redirect_to_profile() throws Exception {
        this.mockMvc.perform(get("/"))
            .andDo(print())
            .andExpect(status().isFound())
            .andExpect(redirectedUrl("/profile"));
    }
}
```

We use `MockMvc` to simulate interactions with a Spring controller without the actual overhead of a Servlet container.

We also use a couple of matchers that Spring provides to assert our result. They actually implement Hamcrest matchers.

The `.andDo(print())` statement will produce a neat debug output for the request and response of the scenario under test. You can comment it if you find it too verbose.

That's all there is to it! The syntax is a bit tricky at the beginning, but an IDE with good completion will be able to help you.

Now we want to test whether, if the user has filled in the test part of their profile, we can redirect them to the correct search. For that, we will need to stub the session with the `MockHttpSession` class:

```
import org.springframework.mock.web.MockHttpSession;
import masterSpringMvc.profile.UserProfileSession;

// put this test below the other one
@Test
public void should_redirect_to_tastes() throws Exception {
    MockHttpSession session = new MockHttpSession();
    UserProfileSession sessionBean = new UserProfileSession();
    sessionBean.setTastes(Arrays.asList("spring", "groovy"));
    session.setAttribute("scopedTarget.userProfileSession",
    sessionBean);

    this.mockMvc.perform(get("/").session(session))
        .andExpect(status().isFound())
        .andExpect(redirectedUrl("/search/mixed;keywords=spring,groo
vy"));
}
```

You will have to add the `setTastes()` setter to the `UserProfileSession` bean for the test to work.

There are a lot of mocking utilities for the Servlet environment in the `org.springframework.mock.web` package.

Note that the attribute representing our bean in session is prefixed by `scopedTarget`. That's because session beans are proxified by Spring. Therefore, there are actually two objects in the Spring context, the actual bean that we defined and its proxy that will end up in the session.

The mock session is a neat class, but we can refactor the test with a builder that will hide implementation details and can be reused later:

```
@Test
public void should_redirect_to_tastes() throws Exception {

    MockHttpSession session = new SessionBuilder()
        .userTastes("spring", "groovy").build();
    this.mockMvc.perform(get("/")
        .session(session)
        .andExpect(status().isFound())
        .andExpect(redirectedUrl("/search/mixed;keywords=spring,groo
vy"));
}
```

The code for the builder is as follows:

```
public class SessionBuilder {
    private final MockHttpSession session;
    UserProfileSession sessionBean;

    public SessionBuilder() {
        session = new MockHttpSession();
        sessionBean = new UserProfileSession();
        session.setAttribute("scopedTarget.userProfileSession",
sessionBean);
    }

    public SessionBuilder userTastes(String... tastes) {
        sessionBean.setTastes(Arrays.asList(tastes));
        return this;
    }

    public MockHttpSession build() {
        return session;
    }
}
```

After this refactoring, your test should always pass, of course.

Mocks and stubs

If we wanted to test the search request handled by the `SearchController` class, we would certainly want to mock `SearchService`.

There are two ways of doing this: with a mock or with a stub.

Mocking with Mockito

First, we can create a mock object with Mockito:

```
package masterSpringMvc.search;

import masterSpringMvc.MasterSpringMvcApplication;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;
import org.springframework.boot.test.SpringApplicationConfiguration;
import org.springframework.test.context.junit4.
SpringJUnit4ClassRunner;
import org.springframework.test.context.web.WebAppConfiguration;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;

import java.util.Arrays;

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

@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = MasterSpringMvcApplication.
class)
@WebAppConfiguration
```

```
public class SearchControllerMockTest {
    @Mock
    private SearchService searchService;

    @InjectMocks
    private SearchController searchController;

    private MockMvc mockMvc;

    @Before
    public void setup() {
        MockitoAnnotations.initMocks(this);
        this.mockMvc = MockMvcBuilders
            .standaloneSetup(searchController)
            .setRemoveSemicolonContent(false)
            .build();
    }

    @Test
    public void should_search() throws Exception {
        when(searchService.search(anyString(), anyListOf(String.class)))
            .thenReturn(Arrays.asList(
                new LightTweet("tweetText")
            ));

        this.mockMvc.perform(get("/search/mixed;keywords=spring"))
            .andExpect(status().isOk())
            .andExpect(view().name("resultPage"))
            .andExpect(model().attribute("tweets", everyItem(
                hasProperty("text", is("tweetText"))
            )));
        verify(searchService, times(1)).search(anyString(),
            anyListOf(String.class));
    }
}
```

You can see that instead of setting up `MockMvc` with the web application context, we have created a standalone context. This context will only contain our controller. That means we have full control over the instantiation and initialization of controllers and their dependencies. It will allow us to easily inject a mock inside of our controller.

The downside is that we have to redeclare pieces of our configuration like the one saying we don't want to remove URL characters after a semicolon.

We use a couple of Hamcrest matchers to assert the properties that will end up in the view model.

The mocking approach has its benefits, such as the ability to verify interactions with the mock and create expectations at runtime.

This will also couple your test with the actual implementation of the object. For instance, if you changed how a tweet is fetched in the controller, you would likely break the tests related to this controller because they still try to mock the service we no longer rely on.

Stubbing our beans while testing

Another approach is to replace the implementation of our `SearchService` class with another one in our test.

We were a bit lazy early on and did not define an interface for `SearchService`. *Always program to an interface and not to an implementation.* Behind this proverbial wisdom lies the most important lesson from the *Gang of Four*.

One of the benefits of the Inversion of Control is to allow for the easy replacement of our implementations in tests or in a real system. For this to work, we will have to modify all the usages `SearchService` with the new interface. With a good IDE, there is a refactoring called `extract interface` that will do just that. This should create an interface that contains the public method `search()` of our `SearchService` class:

```
public interface TwitterSearch {  
    List<LightTweet> search(String searchType, List<String> keywords);  
}
```

Of course, our two controllers, `SearchController` and `SearchApiController`, must now use the interface and not the implementation.

We now have the ability to create a test double for the `TwitterSearch` class specially for our test case. For this to work, we will need to declare a new Spring configuration named `StubTwitterSearchConfig` that will contain another implementation for `TwitterSearch`. I placed it in the `search` package, next to `SearchControllerMockTest`:

```
package masterSpringMvc.search;  
  
import org.springframework.context.annotation.Bean;
```

```
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;

import java.util.Arrays;

@Configuration
public class StubTwitterSearchConfig {
    @Primary @Bean
    public TwitterSearch twitterSearch() {
        return (searchType, keywords) -> Arrays.asList(
            new LightTweet("tweetText"),
            new LightTweet("secondTweet")
        );
    }
}
```

In this configuration class, we redeclare the `TwitterSearch` bean with the `@Primary` annotation, which will tell Spring to use this implementation on priority if other implementations are found in the classpath.

Since the `TwitterSearch` interface contains only one method, we can implement it with a lambda expression.

Here is the complete test that uses our `StubConfiguration` class along with our main configuration with the `SpringApplicationConfiguration` annotation:

```
package masterSpringMvc.search;

import masterSpringMvc.MasterSpringMvcApplication;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.SpringApplicationConfiguration;
import org.springframework.test.context.junit4.
SpringJUnit4ClassRunner;
import org.springframework.test.context.web.WebAppConfiguration;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;
import org.springframework.web.context.WebApplicationContext;

import static org.hamcrest.Matchers.*;
```

```
import static org.springframework.test.web.servlet.request.  
MockMvcRequestBuilders.get;  
import static org.springframework.test.web.servlet.result.  
MockMvcResultMatchers.*;  
  
@RunWith(SpringJUnit4ClassRunner.class)  
@SpringApplicationConfiguration(classes = {  
    MasterSpringMvcApplication.class,  
    StubTwitterSearchConfig.class  
})  
@WebAppConfiguration  
public class SearchControllerTest {  
    @Autowired  
    private WebApplicationContext wac;  
  
    private MockMvc mockMvc;  
  
    @Before  
    public void setup() {  
        this.mockMvc = MockMvcBuilders.webAppContextSetup(this.wac).  
build();  
    }  
  
    @Test  
    public void should_search() throws Exception {  
  
        this.mockMvc.perform(get("/search/mixed;keywords=spring"))  
            .andExpect(status().isOk())  
            .andExpect(view().name("resultPage"))  
            .andExpect(model().attribute("tweets", hasSize(2)))  
            .andExpect(model().attribute("tweets",  
                hasItems(  
                    hasProperty("text",  
                        is("tweetText")),  
                    hasProperty("text",  
                        is("secondTweet"))  
                )));
    }
}
```

Should I use mocks or stubs?

Both approaches have their own merits. For a detailed explanation, check out this great essay by Martin Fowler:
<http://martinfowler.com/articles/mocksArentStubs.html>.

My testing routine is more about writing stubs because I like the idea of testing the output of my objects more than their inner workings. But that's up to you. Spring being a dependency injection framework at its core means that you can easily choose what your favorite approach is.

Unit testing REST controllers

We have just tested a traditional controller redirecting to a view. Testing a REST controller is very similar in principle, but there are a few subtleties.

Since we are going to test the JSON output of our controller, we need a JSON assertion library. Add the following dependency to your build.gradle file:

```
testCompile 'com.jayway.jsonpath:json-path'
```

Let's write a test for the `SearchApiController` class, the controller that allows searching for a tweet and returns results as JSON or XML:

```
package masterSpringMvc.search.api;

import masterSpringMvc.MasterSpringMvcApplication;
import masterSpringMvc.search.StubTwitterSearchConfig;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.SpringApplicationConfiguration;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.
SpringJUnit4ClassRunner;
import org.springframework.test.context.web.WebAppConfiguration;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;
import org.springframework.web.context.WebApplicationContext;
```

```
import static org.hamcrest.Matchers.*;
import static org.springframework.test.web.servlet.request.
MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.
MockMvcResultHandlers.print;
import static org.springframework.test.web.servlet.result.
MockMvcResultMatchers.*;

@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = {
    MasterSpringMvcApplication.class,
    StubTwitterSearchConfig.class
})
@WebAppConfiguration
public class SearchApiControllerTest {
    @Autowired
    private WebApplicationContext wac;

    private MockMvc mockMvc;

    @Before
    public void setup() {
        this.mockMvc = MockMvcBuilders.webAppContextSetup(this.wac) .
build();
    }

    @Test
    public void should_search() throws Exception {

        this.mockMvc.perform(
            get("/api/search/mixed;keywords=spring")
                .accept(MediaType.APPLICATION_JSON) )
            .andDo(print())
            .andExpect(status().isOk())
            .andExpect(content().contentTypeCompatibleWith(MediaType.
APPLICATION_JSON))
            .andExpect(jsonPath("$", hasSize(2)))
            .andExpect(jsonPath("$.text", is("tweetText")))
            .andExpect(jsonPath("$.text", is("secondTweet")));
    }
}
```

Note the simple and elegant assertions on the JSON output. Testing our user controller will require a bit more work.

First, let's add `assertj` to the classpath; it will help us write cleaner tests:

```
testCompile 'org.assertj:assertj-core:3.0.0'
```

Then, to simplify testing, add a `reset()` method to our `UserRepository` class that will help us with the test:

```
void reset(User... users) {  
    userMap.clear();  
    for (User user : users) {  
        save(user);  
    }  
}
```

In real life, we should probably extract an interface and create a stub for testing. I will leave that as an exercise for you.

Here is the first test that gets the list of users:

```
package masterSpringMvc.user.api;  
  
import masterSpringMvc.MasterSpringMvcApplication;  
import masterSpringMvc.user.User;  
import masterSpringMvc.user.UserRepository;  
import org.junit.Before;  
import org.junit.Test;  
import org.junit.runner.RunWith;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.test.SpringApplicationConfiguration;  
import org.springframework.http.MediaType;  
import org.springframework.test.context.junit4.  
SpringJUnit4ClassRunner;  
import org.springframework.test.context.web.WebAppConfiguration;  
import org.springframework.test.web.servlet.MockMvc;  
import org.springframework.test.web.servlet.setup.MockMvcBuilders;  
import org.springframework.web.context.WebApplicationContext;  
  
import static org.hamcrest.Matchers.*;  
import static org.springframework.test.web.servlet.request.  
MockMvcRequestBuilders.*;  
import static org.springframework.test.web.servlet.result.  
MockMvcResultMatchers.*;
```

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = MasterSpringMvcApplication.
class)
@WebAppConfiguration
public class UserApiControllerTest {

    @Autowired
    private WebApplicationContext wac;

    @Autowired
    private UserRepository userRepository;

    private MockMvc mockMvc;

    @Before
    public void setup() {
        this.mockMvc = MockMvcBuilders.webAppContextSetup(this.wac) .
build();
        userRepository.reset(new User("bob@spring.io"));
    }

    @Test
    public void should_list_users() throws Exception {
        this.mockMvc.perform(
            get("/api/users")
                .accept(MediaType.APPLICATION_JSON)
        )
            .andExpect(status().isOk())
            .andExpect(content().contentTypeCompatibleWith(MediaType.
APPLICATION_JSON))
            .andExpect(jsonPath("$", hasSize(1)))
            .andExpect(jsonPath("$.[0].email", is("bob@spring.
io")));
    }
}
```

For this to work, add a constructor to the `User` class, taking the e-mail property as a parameter. Be careful: you also need to have a default constructor for Jackson.

The test is very similar to the previous test with the additional setup of `UserRepository`.

Let's test the POST method that creates a user now:

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

// Insert this test below the previous one
@Test
public void should_create_new_user() throws Exception {
    User user = new User("john@spring.io");
    this.mockMvc.perform(
        post("/api/users")
            .contentType(MediaType.APPLICATION_JSON)
            .content(JsonUtil.toJson(user))
    )
        .andExpect(status().isCreated());

    assertThat(userRepository.findAll())
        .extracting(User::getEmail)
        .containsOnly("bob@spring.io", "john@spring.io");
}
```

There are two things to be noted. The first one is the use of AssertJ to assert the content of the repository after the test. You will need the following static import for that to work:

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

The second is that we use a utility method to convert our object to JSON before sending it to the controller. For that purpose, I created a simple utility class in the `utils` package, as follows:

```
package masterSpringMvc.utils;

import com.fasterxml.jackson.annotation.JsonInclude;
import com.fasterxml.jackson.databind.ObjectMapper;

import java.io.IOException;

public class JsonUtil {
    public static byte[] toJson(Object object) throws IOException {
        ObjectMapper mapper = new ObjectMapper();
        mapper.setSerializationInclusion(JsonInclude.Include.NON_NULL);
        return mapper.writeValueAsBytes(object);
    }
}
```

The tests for the `DELETE` method are as follows:

```
@Test
public void should_delete_user() throws Exception {
    this.mockMvc.perform(
        delete("/api/user/bob@spring.io")
            .accept(MediaType.APPLICATION_JSON)
    )
        .andExpect(status().isOk());

    assertThat(userRepository.findAll()).hasSize(0);
}

@Test
public void should_return_not_found_when_deleting_unknown_user()
throws Exception {
    this.mockMvc.perform(
        delete("/api/user/non-existing@mail.com")
            .accept(MediaType.APPLICATION_JSON)
    )
        .andExpect(status().isNotFound());
}
```

Finally, here's the test for the `PUT` method, which updates a user:

```
@Test
public void put_should_update_existing_user() throws Exception {
    User user = new User("ignored@spring.io");
    this.mockMvc.perform(
        put("/api/user/bob@spring.io")
            .content(JsonUtil.toJson(user))
            .contentType(MediaType.APPLICATION_JSON)
    )
        .andExpect(status().isOk());

    assertThat(userRepository.findAll())
        .extracting(User::getEmail)
        .containsOnly("bob@spring.io");
}
```

Whoops! The last test does not pass! By checking the implementation of `UserApiController`, we can easily see why:

```
    @RequestMapping(value = "/user/{email}", method = RequestMethod.PUT)
    public ResponseEntity<User> updateUser(@PathVariable String email,
    @RequestBody User user) throws EntityNotFoundException {
        User saved = userRepository.update(email, user);
        return new ResponseEntity<>(saved, HttpStatus.CREATED);
    }
```

We returned the wrong status in the controller! Change it to `HttpStatus.OK` and the test should be green again.

With Spring, one can easily write controller tests using the same configuration of our application, but we can just as efficiently override or change some elements in our testing setup.

Another interesting thing that you will notice while running all the tests is that the application context is only loaded once, which means that the overhead is actually very small.

Our application is small too, so we did not make any effort to split our configuration into reusable chunks. It can be a really good practice not to load the full application context inside of every test. You can actually split the component scanned into different units with the `@ComponentScan` annotation.

This annotation has several attributes that allow you to define filters with `includeFilter` and `excludeFilter` (loading only the controller for instance) and scan specific packages with the `basePackageClasses` and `basePackages` annotations.

You can also split your configuration into multiple `@Configuration` classes. A good example would be splitting the code for the users and for the tweet parts of our application into two independent parts.

We will now have a look at acceptance tests, which are a very different kind of beast.

Testing the authentication

If you wish to set up Spring Security in a MockMvc test, you can write this test next to our previous test:

```
package masterSpringMvc.user.api;

import masterSpringMvc.MasterSpringMvcApplication;
import masterSpringMvc.user.User;
import masterSpringMvc.user.UserRepository;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.SpringApplicationConfiguration;
import org.springframework.http.MediaType;
import org.springframework.security.web.FilterChainProxy;
import org.springframework.test.context.junit4.
SpringJUnit4ClassRunner;
import org.springframework.test.context.web.WebAppConfiguration;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;
import org.springframework.web.context.WebApplicationContext;

import java.util.Base64;

import static org.springframework.test.web.servlet.request.
MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.
MockMvcResultMatchers.status;

@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = MasterSpringMvcApplication.
class)
@WebAppConfiguration
public class UserApiControllerAuthTest {

    @Autowired
    private FilterChainProxy springSecurityFilter;
```

```
    @Autowired
    private WebApplicationContext wac;

    @Autowired
    private UserRepository userRepository;

    private MockMvc mockMvc;

    @Before
    public void setup() {
        this.mockMvc = MockMvcBuilders.webAppContextSetup(this.wac) .
        addFilter(springSecurityFilter).build();
        userRepository.reset(new User("bob@spring.io"));
    }

    @Test
    public void unauthenticated_cannot_list_users() throws Exception {
        this.mockMvc.perform(
            get("/api/users")
                .accept(MediaType.APPLICATION_JSON)
        )
            .andExpect(status().isUnauthorized());
    }

    @Test
    public void admin_can_list_users() throws Exception {
        this.mockMvc.perform(
            get("/api/users")
                .accept(MediaType.APPLICATION_JSON)
                .header("Authorization", basicAuth("admin",
"admin"))
        )
            .andExpect(status().isOk());
    }

    private String basicAuth(String login, String password) {
        byte[] auth = (login + ":" + password).getBytes();
        return "Basic " + Base64.getEncoder().encodeToString(auth);
    }
}
```

In the preceding example, we added `springSecurityFilter` to our configuration. This will activate Spring Security checks. To test if the authentication works, we simply send the correct headers along with the request we would like to perform.

The advantage of basic authentication is that it's really straightforward to simulate. With a more complicated setup, you would have to perform a mock request on the authentication endpoint.

At the time of writing, Spring Boot is at version 1.2.3 and depends on Spring Security 3.

In a few weeks, Spring Boot 1.3.0 will be available, it will update Spring Security and use version 4.

This is good news because Spring Security 4 includes a really easy setup of the authenticated user with simple annotations. See <http://docs.spring.io/spring-security/site/docs/4.0.x/reference/htmlsingle/#test> for more details.

Writing acceptance tests

Unit tests can only cover a subset of the different interactions between the components of our application. To go a little further, we will need to set up acceptance tests, tests that will actually boot up the complete application and allow us to interact with its interface.

The Gradle configuration

The first thing we will want to do when we add integration tests to a project is to put them in a different location to that of the unit tests.

The reason for this is, essentially, that acceptance tests are slower than unit tests. They can be part of a different integration job, such as a nightly build, and we want developers to be able to launch the different kinds of tests easily from their IDE. To do this with Gradle, we will have to add a new configuration called `integrationTest`. For Gradle, a configuration is a group of artifacts and their dependencies. We already have several configurations in our project: `compile`, `testCompile`, and so on.

You can have a look at the configurations of your project, and much more, by typing `./gradlew properties` at the root of your project.

Add a new configuration at the end of `build.gradle` file:

```
configurations {  
    integrationTestCompile.extendsFrom testCompile  
    integrationTestRuntime.extendsFrom testRuntime  
}
```

This will allow you to declare dependencies for `integrationTestCompile` and `integrationTestRuntime`. More importantly, by inheriting the test configurations, we have access to their dependencies.



I do not recommend declaring your integration test dependencies as `integrationTestCompile`. It will work as far as Gradle is concerned, but support inside of IDE is non-existent. What I usually do is declare my integration test dependencies as `testCompile` dependencies instead. This is only a small inconvenience.

Now that we have our new configurations, we must create a `sourceSet` class associated with them. A `sourceSet` class represents a logical group of Java source and resources. Naturally, they also have to inherit from the test and main classes; see the following code:

```
sourceSets {  
    integrationTest {  
        compileClasspath += main.output + test.output  
        runtimeClasspath += main.output + test.output  
    }  
}
```

Finally, we need to add a task to run them from our build, as follows:

```
task integrationTest(type: Test) {  
    testClassesDir = sourceSets.integrationTest.output.classesDir  
    classpath = sourceSets.integrationTest.runtimeClasspath  
    reports.html.destination = file("${reporting.baseDir}/  
integrationTests")  
}
```

To run our test, we can type `./gradlew integrationTest`. Besides configuring our classpath and where to find our test classes, we also defined a directory where the test report will be generated.

This configuration allows us to write our tests in `src/integrationTest/java` or `src/integrationTest/groovy`, which will make it easier to identify them and run them separately from our unit tests.

By default, they will be generated in `build/reports/tests`. If we do not override them, if we launch both tests and integration tests with `gradle clean test integrationTest`, they will override each other.

It's also worth mentioning that a young plugin in the Gradle ecosystem aims to simplify declaring new test configurations, visit <https://plugins.gradle.org/plugin/org.unbroken-dome.test-sets> for detailed information.

Our first FluentLenium test

FluentLenium is an amazing library for piloting Selenium tests. Let's add a few dependencies to our build script:

```
testCompile 'org.fluentlenium:fluentlenium-assertj:0.10.3'  
testCompile 'com.codeborne:phantomjsdriver:1.2.1'  
testCompile 'org.seleniumhq.selenium:selenium-java:2.45.0'
```

By default, `fluentlenium` comes with `selenium-java`. We redeclare it just to explicitly require the latest version available. We also added a dependency to the `PhantomJS` driver, which is not officially supported by Selenium. The problem with the `selenium-java` library is that it comes bundled with all the supported web drivers.

You can see the dependency tree of our project by typing `gradle dependencies`. At the bottom, you will see something like this:

```
+--- org.fluentlenium:fluentlenium-assertj:0.10.3  
|    +--- org.fluentlenium:fluentlenium-core:0.10.3  
|    |    \--- org.seleniumhq.selenium:selenium-java:2.44.0 -> 2.45.0  
|    |    +--- org.seleniumhq.selenium:selenium-chrome-  
driver:2.45.0  
  
|    |    +--- org.seleniumhq.selenium:selenium-htmlunit-  
driver:2.45.0  
  
|    |    +--- org.seleniumhq.selenium:selenium-firefox-  
driver:2.45.0  
  
|    |    +--- org.seleniumhq.selenium:selenium-ie-driver:2.45.0  
  
|    |    +--- org.seleniumhq.selenium:selenium-safari-  
driver:2.45.0  
  
|    |    +--- org.webbitserver:webbit:0.4.14 (*)  
|    |    \--- org.seleniumhq.selenium:selenium-leg-rc:2.45.0  
|    |    \--- org.seleniumhq.selenium:selenium-remote-  
driver:2.45.0 (*)  
|    \--- org.assertj:assertj-core:1.6.1 -> 3.0.0
```

Having all those dependencies in the classpath is highly unnecessary since we will just use the PhantomJS driver. To exclude the dependencies we won't need, we can add the following part to our buildscript, right before the dependencies declaration:

```
configurations {
    testCompile {
        exclude module: 'selenium-safari-driver'
        exclude module: 'selenium-ie-driver'
        //exclude module: 'selenium-firefox-driver'
        exclude module: 'selenium-htmlunit-driver'
        exclude module: 'selenium-chrome-driver'
    }
}
```

We just keep the `firefox` driver at hand. `PhantomJS` driver is a headless browser, so understanding what happens without a GUI can prove tricky. It can be nice to switch to Firefox to debug a complex test.

With our classpath correctly configured, we can now write our first integration test. Spring Boot has a very convenient annotation to support this test:

```
import masterSpringMvc.MasterSpringMvcApplication;
import masterSpringMvc.search.StubTwitterSearchConfig;
import org.fluentlenium.adapter.FluentTest;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.phantomjs.PhantomJSDriver;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.test.SpringApplicationConfiguration;
import org.springframework.boot.test.WebIntegrationTest;
import org.springframework.test.context.junit4.
SpringJUnit4ClassRunner;

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

@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = {
    MasterSpringMvcApplication.class,
    StubTwitterSearchConfig.class
})
@WebIntegrationTest(randomPort = true)
public class FluentIntegrationTest extends FluentTest {
```

```

    @Value("${local.server.port}")
    private int serverPort;

    @Override
    public WebDriver getDefaultDriver() {
        return new PhantomJSDriver();
    }

    public String getDefaultBaseUrl() {
        return "http://localhost:" + serverPort;
    }

    @Test
    public void hasPageTitle() {
        goTo("/");
        assertThat(findFirst("h2").getText()).isEqualTo("Login");
    }
}

```

Note that FluentLenium has a neat API for requesting DOM elements. With AssertJ, we can then write easy-to read-assertions on the page content.



Have a look at the documentation at
<https://github.com/FluentLenium/FluentLenium>
 for further information.

With the `@WebIntegrationTest` annotation, Spring will actually create the embedded Servlet container (Tomcat) and launch our web application on a random port! We need to retrieve this port number at runtime. This will allow us to provide a base URL for our tests, a URL that will be the prefix for all the navigation we do in our tests.

If you try to run the test at this stage, you will see the following error message:

```
java.lang.IllegalStateException: The path to the driver executable must
be set by the phantomjs.binary.path capability/system property/PATH
variable; for more information, see https://github.com/ariya/phantomjs/
wiki. The latest version can be downloaded from http://phantomjs.org/
download.html
```

Indeed, PhantomJS needs to be installed on your machine for this to work correctly. On a Mac, simply use `brew install phantomjs`. For other platforms, see the documentation at <http://phantomjs.org/download.html>.

If you don't want to install a new binary on your machine, replace new `PhantomJSdriver()` with new `FirefoxDriver()`. Your test will be a bit slower, but you will have a GUI.

Our first test is landing on the profile page, right? We need to find a way to log in now.

What about faking login with a stub?

Put this class in the test sources (`src/test/java`):

```
package masterSpringMvc.auth;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;
import org.springframework.security.authentication.
UsernamePasswordAuthenticationToken;
import org.springframework.security.core.context.
SecurityContextHolder;
import org.springframework.social.connect.ConnectionFactoryLocator;
import org.springframework.social.connect.UsersConnectionRepository;
import org.springframework.social.connect.web.
ProviderSignInController;
import org.springframework.social.connect.web.SignInAdapter;
import org.springframework.web.context.request.NativeWebRequest;
import org.springframework.web.servlet.view.RedirectView;

@Configuration
public class StubSocialSigninConfig {

    @Bean
    @Primary
    @Autowired
    public ProviderSignInController signInController(ConnectionFactory
Locator factoryLocator,
UsersConnectionRepository usersRepository,
                                                SignInAdapter
signInAdapter) {
        return new FakeSigninController(factoryLocator,
usersRepository, signInAdapter);
    }
}
```

```

public class FakeSigninController extends ProviderSignInController
{
    public FakeSigninController(ConnectionFactoryLocator
connectionFactoryLocator,
                               UsersConnectionRepository
usersConnectionRepository,
                               SignInAdapter signInAdapter) {
        super(connectionFactoryLocator, usersConnectionRepository,
signInAdapter);
    }

    @Override
    public RedirectView signIn(String providerId, NativeWebRequest
request) {
        UsernamePasswordAuthenticationToken authentication =
new UsernamePasswordAuthenticationToken("geowar
in", null, null);
        SecurityContextHolder.getContext().setAuthentication(auth
entication);
        return new RedirectView("/");
    }
}
}

```

This will authenticate any user clicking on the Twitter sign in button as geowarin.

We will write a second test that will fill the profile form and assert that the search result is displayed:

```

import masterSpringMvc.MasterSpringMvcApplication;
import masterSpringMvc.auth.StubSocialSigninConfig;
import masterSpringMvc.search.StubTwitterSearchConfig;
import org.fluentlenium.adapter.FluentTest;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.phantomjs.PhantomJSDriver;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.test.SpringApplicationConfiguration;
import org.springframework.boot.test.WebIntegrationTest;
import org.springframework.test.context.junit4.
SpringJUnit4ClassRunner;

import static org.assertj.core.api.Assertions.assertThat;
import static org.fluentlenium.core.filter.FilterConstructor.withName;

```

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest(classes = {
    MasterSpringMvcApplication.class,
    StubTwitterSearchConfig.class,
    StubSocialSigninConfig.class
})
@WebIntegrationTest(randomPort = true)
public class FluentIntegrationTest extends FluentTest {

    @Value("${local.server.port}")
    private int serverPort;

    @Override
    public WebDriver getDefaultDriver() {
        return new PhantomJSdriver();
    }

    public String getDefaultBaseUrl() {
        return "http://localhost:" + serverPort;
    }

    @Test
    public void hasPageTitle() {
        goTo("/");
        assertThat(findFirst("h2").getText()).isEqualTo("Login");
    }

    @Test
    public void should_be_redirected_after_filling_form() {
        goTo("/");
        assertThat(findFirst("h2").getText()).isEqualTo("Login");

        find("button", withName("twitterSignin")).click();
        assertThat(findFirst("h2").getText()).isEqualTo("Your
profile");

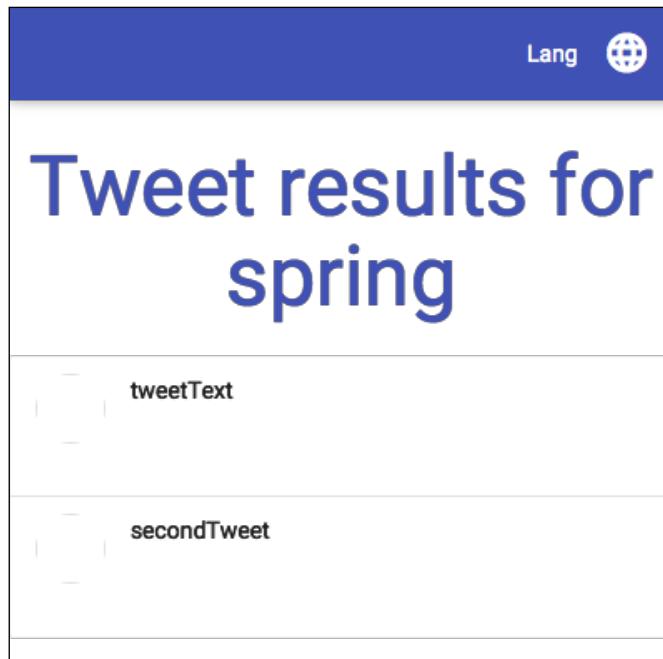
        fill("#twitterHandle").with("geowarin");
        fill("#email").with("geowarin@mymail.com");
        fill("#birthDate").with("03/19/1987");

        find("button", withName("addTaste")).click();
        fill("#tastes0").with("spring");
    }
}
```

```
        find("button", withName("save")).click();

        takeScreenShot();
        assertThat(findFirst("h2").getText()).isEqualTo("Tweet results
for spring");
        assertThat(findFirst("ul.collection").find("li")).hasSize(2);
    }
}
```

Note that we can easily ask our web driver to take a screenshot of the current browser used for testing. This will produce the following output:



Page Objects with FluentLenium

The previous test was a bit messy. We have hardcoded all the selectors in our test. This can become very risky when we write a lot of tests using the same elements because whenever we change the page layout, all the tests will break. Moreover, the test is a little difficult to read.

To fix this, a common practice is to use a page object that will represent a specific web page in our application. With FluentLenium, page objects must inherit the `FluentPage` class.

We will create three pages, one for each element of our GUI. The first one will be the login page with the option to click on the `twitterSignin` button, the second one will be the profile page with convenience methods for filling in the profile form, and the last one will be the result page on which we can assert the results displayed.

Let's create the login page at once. I put all the three pages in a `pages` package:

```
package pages;

import org.fluentlenium.core.FluentPage;
import org.fluentlenium.core.domain.FluentWebElement;
import org.openqa.selenium.support.FindBy;

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

public class LoginPage extends FluentPage {
    @FindBy(name = "twitterSignin")
    FluentWebElement signinButton;

    public String getUrl() {
        return "/login";
    }

    public void isAt() {
        assertThat(findFirst("h2").getText()).isEqualTo("Login");
    }

    public void login() {
        signinButton.click();
    }
}
```

Let's create one page for our profile page:

```
package pages;

import org.fluentlenium.core.FluentPage;
import org.fluentlenium.core.domain.FluentWebElement;
import org.openqa.selenium.support.FindBy;

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

public class ProfilePage extends FluentPage {
    @FindBy(name = "addTaste")
```

```
FluentWebElement addTasteButton;
@FindBy(name = "save")
FluentWebElement saveButton;

public String getUrl() {
    return "/profile";
}

public void isAt() {
    assertThat(findFirst("h2").getText()).isEqualTo("Your
profile");
}

public void fillInfos(String twitterHandle, String email, String
birthDate) {
    fill("#twitterHandle").with(twitterHandle);
    fill("#email").with(email);
    fill("#birthDate").with(birthDate);
}

public void addTaste(String taste) {
    addTasteButton.click();
    fill("#tastes0").with(taste);
}

public void saveProfile() {
    saveButton.click();
}
}
```

Let's also create another one for the search result page:

```
package pages;

import com.google.common.base.Joiner;
import org.fluentlenium.core.FluentPage;
import org.fluentlenium.core.domain.FluentWebElement;
import org.openqa.selenium.support.FindBy;

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

public class SearchResultPage extends FluentPage {
    @FindBy(css = "ul.collection")
    FluentWebElement resultList;
```

```
public void isAt(String... keywords) {
    assertThat(findFirst("h2").getText())
        .isEqualTo("Tweet results for " + Joiner.on(",") .
join(keywords));
}

public int getNumberOfResults() {
    return resultList.find("li").size();
}
}
```

We can now refactor the test using those Page Objects:

```
@Page
private LoginPage loginPage;
@Page
private ProfilePage profilePage;
@Page
private SearchResultPage searchResultPage;

@Test
public void should_be_redirected_after_filling_form() {
    goTo("/");
    loginPage.isAt();

    loginPage.login();
    profilePage.isAt();

    profilePage.fillInfos("geowarin", "geowarin@mymail.com",
"03/19/1987");
    profilePage.addTaste("spring");

    profilePage.saveProfile();

    takeScreenShot();
    searchResultPage.isAt();
    assertThat(searchResultPage.getNumberOfResults()).isEqualTo(2);
}
```

Much more readable, isn't it?

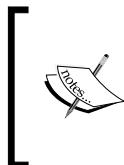
Making our tests more Groovy

If you don't know Groovy, consider it like a close cousin of Java, without the verbosity. Groovy is a dynamic language with optional typing. This means that you can have the guarantees of a type system when it matters and the versatility of duck typing when you know what you are doing.

With this language, you can write POJOs without getters, setters, `equals` and `hashCode` methods. Everything is handled for you.

Writing `==` will actually call the `equals` method. The operators can be overloaded, which allows a neat syntax with little arrows, such as `<<`, to write text to a file, for instance. It also means that you can add integers to `BIGINTegers` and get a correct result.

The **Groovy Development Kit (GDK)** also adds several very interesting methods to classic Java objects. It also considers regular expressions and closures as first-class citizens.



If you want a solid introduction to Groovy, check out the Groovy style guide at <http://www.groovy-lang.org/style-guide.html>.

You can also watch this amazing presentation by Peter Ledbrook at <http://www.infoq.com/presentations/groovy-for-java>.

As far as I am concerned, I always try to push Groovy on the testing side of the application I work on. It really improves the readability of the code and the productivity of developers.

Unit tests with Spock

To be able to write Groovy tests in our project, we need to use the Groovy plugin instead of the Java plugin.

Here's what you have in your build script:

```
apply plugin: 'java'
```

Change it to the following:

```
apply plugin: 'groovy'
```

This modification is perfectly harmless. The Groovy plugin extends the Java plugin, so the only difference it makes is that it gives the ability to add Groovy source in `src/main/groovy`, `src/test/groovy` and `src/integrationTest/groovy`.

Obviously, we also need to add Groovy to the classpath. We will also add Spock, the most popular Groovy testing library, via the `spock-spring` dependency, which will enable compatibility with Spring:

```
testCompile 'org.codehaus.groovy:groovy-all:2.4.4:indy'  
testCompile 'org.spockframework:spock-spring'
```

We can now rewrite `HomeControllerTest` with a different approach. Let's create a `HomeControllerSpec` class in `src/test/groovy`. I added it to the `masterSpringMvc.controller` package just like our first instance of `HomeControllerTest`:

```
package masterSpringMvc.controller  
  
import masterSpringMvc.MasterSpringMvcApplication  
import masterSpringMvc.search.StubTwitterSearchConfig  
import org.springframework.beans.factory.annotation.Autowired  
import org.springframework.boot.test.SpringApplicationContextLoader  
import org.springframework.test.context.ContextConfiguration  
import org.springframework.test.context.web.WebAppConfiguration  
import org.springframework.test.web.servlet.MockMvc  
import org.springframework.test.web.servlet.setup.MockMvcBuilders  
import org.springframework.web.context.WebApplicationContext  
import spock.lang.Specification  
  
import static org.springframework.test.web.servlet.request.  
MockMvcRequestBuilders.*;  
import static org.springframework.test.web.servlet.result.  
MockMvcResultMatchers.*;  
  
@ContextConfiguration(loader = SpringApplicationContextLoader,  
                     classes = [MasterSpringMvcApplication,  
StubTwitterSearchConfig])  
@WebAppConfiguration  
class HomeControllerSpec extends Specification {  
    @Autowired  
    WebApplicationContext wac;  
  
    MockMvc mockMvc;  
  
    def setup() {  
        mockMvc = MockMvcBuilders.webAppContextSetup(this.wac).  
build();  
    }  
}
```

```
def "User is redirected to its profile on his first visit"() {  
    when: "I navigate to the home page"  
    def response = this.mockMvc.perform(get("/"))  
  
    then: "I am redirected to the profile page"  
    response  
        .andExpect(status().isFound())  
        .andExpect(redirectedUrl("/profile"))  
    }  
}
```

Our test instantaneously became more readable with the ability to use strings as method names and the little BDD DSL (Domain Specific Language) provided by Spock. This is not directly visible here, but every statement inside of a `then` block will implicitly be an assertion.

At the time of writing, because Spock doesn't read meta annotations, the `@SpringApplicationConfiguration` annotation cannot be used so we just replaced it with `@ContextConfiguration(loader = SpringApplicationContextLoader)`, which is essentially the same thing.

We now have two versions of the same test, one in Java and the other in Groovy. It is up to you to choose the one that best fits your style of coding and remove the other one. If you decide to stick with Groovy, you will have to rewrite the `should_redirect_to_tastes()` test in Groovy. It should be easy enough.

Spock also has powerful support for mocks. We can rewrite the previous `SearchControllerMockTest` class a bit differently:

```
package masterSpringMvc.search  
  
import masterSpringMvc.MasterSpringMvcApplication  
import org.springframework.boot.test.SpringApplicationContextLoader  
import org.springframework.test.context.ContextConfiguration  
import org.springframework.test.context.web.WebAppConfiguration  
import org.springframework.test.web.servlet.setup.MockMvcBuilders  
import spock.lang.Specification  
  
import static org.hamcrest.Matchers.*;  
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;  
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;
```

```
@ContextConfiguration(loader = SpringApplicationContextLoader,
    classes = [MasterSpringMvcApplication])
@WebAppConfiguration
class SearchControllerMockSpec extends Specification {
    def twitterSearch = Mock(TwitterSearch)
    def searchController = new SearchController(twitterSearch)

    def mockMvc = MockMvcBuilders.standaloneSetup(searchController)
        .setRemoveSemicolonContent(false)
        .build()

    def "searching for the spring keyword should display the search
page"() {
        when: "I search for spring"
        def response = mockMvc.perform(get("/search/
mixed;keywords=spring"))

        then: "The search service is called once"
        1 * twitterSearch.search(_, _) >> [new
LightTweet('tweetText')]

        and: "The result page is shown"
        response
            .andExpect(status().isOk())
            .andExpect(view().name("resultPage"))

        and: "The model contains the result tweets"
        response
            .andExpect(model().attribute("tweets", everyItem(
                hasProperty("text", is("tweetText"))))
        )))
    }
}
```

All the verbosity of Mockito is now gone. The then block actually asserts that the twitterSearch method is called once (1 *) with any parameter (_, _). Like with Mockito, we could have expected specific parameters.

The double arrow >> syntax is used to return an object from the mocked method. In our case, it's a list containing only one element.

With only a little dependency in our classpath, we have already written more readable tests, but we're not done yet. We will also refactor our acceptance tests to use Geb, a Groovy library that pilots Selenium tests.

Integration tests with Geb

Geb is the de facto library for writing tests in the Grails framework. Although its version is 0.12.0, it is very stable and extremely comfortable to work with.

It provides a selector API à la jQuery, which makes tests easy to write, even for frontend developers. Groovy is also a language that has some JavaScript influences that will also appeal to them.

Let's add Geb with the support for Spock specifications to our classpath:

```
testCompile 'org.gebish:geb-spock:0.12.0'
```

Geb can be configured via a Groovy script found directly at the root of `src/integrationTest/groovy`, called `GebConfig.groovy`:

```
import org.openqa.selenium.Dimension
import org.openqa.selenium.firefox.FirefoxDriver
import org.openqa.selenium.phantomjs.PhantomJSDriver

reportsDir = new File('./build/geb-reports')
driver = {
    def driver = new FirefoxDriver()
    // def driver = new PhantomJSDriver()
    driver.manage().window().setSize(new Dimension(1024, 768))
    return driver
}
```

In this configuration, we indicate where Geb will generate its reports and which driver to use. Reports in Geb are an enhanced version of screenshots, which also contains the current page in HTML. Their generation can be triggered at any moment by calling the `report` function inside a Geb test.

Let's rewrite our first integration test with Geb:

```
import geb.Configuration
import geb.spock.GebSpec
import masterSpringMvc.MasterSpringMvcApplication
import masterSpringMvc.search.StubTwitterSearchConfig
import org.springframework.beans.factory.annotation.Value
import org.springframework.boot.test.SpringApplicationContextLoader
import org.springframework.boot.test.WebIntegrationTest
import org.springframework.test.context.ContextConfiguration
```

```
@ContextConfiguration(loader = SpringApplicationContextLoader,
    classes = [MasterSpringMvcApplication,
    StubTwitterSearchConfig])
@WebIntegrationTest(randomPort = true)
class IntegrationSpec extends GebSpec {

    @Value('${local.server.port}')
    int port

    Configuration createConf() {
        def configuration = super.createConf()
        configuration.baseUrl = "http://localhost:$port"
        configuration
    }

    def "User is redirected to the login page when not logged"() {
        when: "I navigate to the home page"
        go '/'
        //      report 'navigation-redirection'

        then: "I am redirected to the profile page"
        $('h2', 0).text() == 'Login'
    }
}
```

For the moment, it is very similar to FluentLenium. We can already see the \$ function, which will allow us to grab a DOM element via its selector. Here, we also state that we want the first h2 in the page by giving the 0 index.

Page Objects with Geb

Page objects with Geb are a real pleasure to work with. We will create the same page objects that we did previously so that you can appreciate the differences.

With Geb, the Page Objects must inherit from the geb.Page class. First, let's create the LoginPage. I suggest avoiding putting it in the same package as the previous one. I created a package called geb.pages:

```
package geb.pages

import geb.Page

class LoginPage extends Page {
```

```
static url = '/login'
static at = { $('h2', 0).text() == 'Login' }
static content = {
    twitterSignin { $('button', name: 'twitterSignin') }
}

void loginWithTwitter() {
    twitterSignin.click()
}
}
```

Then we can create the ProfilePage:

```
package geb.pages

import geb.Page

class ProfilePage extends Page {

    static url = '/profile'
    static at = { $('h2', 0).text() == 'Your profile' }
    static content = {
        addTasteButton { $('button', name: 'addTaste') }
        saveButton { $('button', name: 'save') }
    }

    void fillInfos(String twitterHandle, String email, String
birthDate) {
        $("#twitterHandle") << twitterHandle
        $("#email") << email
        $("#birthDate") << birthDate
    }

    void addTaste(String taste) {
        addTasteButton.click()
        $("#tastes0") << taste
    }

    void saveProfile() {
        saveButton.click();
    }
}
```

This is basically the same page as before. Note the little `<<` to assign values to an input element. You could also call `setText` on them.

The `at` method is completely part of the framework, and Geb will automatically assert those when you navigate to the corresponding page.

Let's create the `SearchResultPage`:

```
package geb.pages

import geb.Page

class SearchResultPage extends Page {
    static url = '/search'
    static at = { $('h2', 0).text().startsWith('Tweet results for') }
    static content = {
        resultList { $('ul.collection') }
        results { resultList.find('li') }
    }
}
```

It's a bit shorter, thanks to the ability to reuse previously defined content for the results.

With out the Page Object set up, we can write the test as follows:

```
import geb.Configuration
import geb.pages.LoginPage
import geb.pages.ProfilePage
import geb.pages.SearchResultPage
import geb.spock.GebSpec
import masterSpringMvc.MasterSpringMvcApplication
import masterSpringMvc.auth.StubSocialSigninConfig
import masterSpringMvc.search.StubTwitterSearchConfig
import org.springframework.beans.factory.annotation.Value
import org.springframework.boot.test.SpringApplicationContextLoader
import org.springframework.boot.test.WebIntegrationTest
import org.springframework.test.context.ContextConfiguration

@ContextConfiguration(loader = SpringApplicationContextLoader,
    classes = [MasterSpringMvcApplication,
    StubTwitterSearchConfig, StubSocialSigninConfig])
@WebIntegrationTest(randomPort = true)
class IntegrationSpec extends GebSpec {

    @Value('${local.server.port}')
    int port
```

```
Configuration createConf() {
    def configuration = super.createConf()
    configuration.baseUrl = "http://localhost:$port"
    configuration
}

def "User is redirected to the login page when not logged"() {
    when: "I navigate to the home page"
    go '/'

    then: "I am redirected to the login page"
    $('h2').text() == 'Login'
}

def "User is redirected to its profile on his first visit"() {
    when: 'I am connected'
    to LoginPage
    loginWithTwitter()

    and: "I navigate to the home page"
    go '/'

    then: "I am redirected to the profile page"
    $('h2').text() == 'Your profile'
}

def "After filling his profile, the user is taken to result
matching his tastes"() {
    given: 'I am connected'
    to LoginPage
    loginWithTwitter()

    and: 'I am on my profile'
    to ProfilePage

    when: 'I fill my profile'
    fillInfos("geowarin", "geowarin@mymail.com", "03/19/1987");
    addTaste("spring")

    and: 'I save it'
    saveProfile()
```

```
        then: 'I am taken to the search result page'
        at SearchResultPage
        page.results.size() == 2
    }
}
```

My, what a beauty! You can certainly write your user stories directly with Geb!

With our simple tests, we only scratched the surface of Geb. There is much more functionality available, and I encourage you to read the *Book of Geb*, a very fine piece of documentation available at <http://www.gebish.org/manual/current/>.

The check point

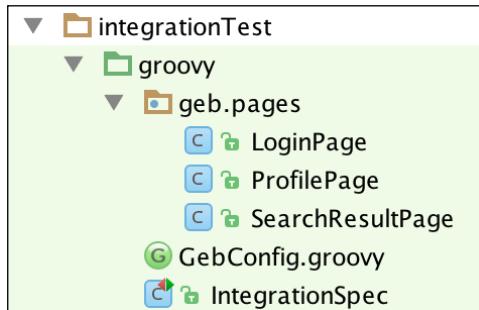
In this chapter, we added a bunch of tests in `src/test/java`. I chose to go with Groovy, so I deleted the duplicated tests:



In the `src/test/groovy` directory, I have refactored two tests as follows:



In `src/integrationTest/groovy`, we have an integration test written with Geb:



Finally, we added an `integrationTest` task to the Gradle build. Run `gradle clean test` and `gradle clean integrationTest` to make sure that all your tests pass.

If the build is successful, we are ready for the next chapter.

Summary

In this chapter, we've studied the differences between unit and integration tests.

We saw how testing is a healthy habit that will give us confidence in what we build and what we ship. It will save us money and spare some headaches in the long run.

Spring works well with classical JUnit tests written in Java, and it has first-class support for integration tests. But we can also easily use other languages, such as Groovy, to make the tests more readable and easier to write.

Testing is undeniably one of the strongest points of the Spring framework and one of the main reasons to use dependency injection in the first place.

Stay tuned for the next chapter, where we will optimize our application so that it is ready to be deployed in the cloud!

8

Optimizing Your Requests

In this chapter, we will be looking at different techniques to improve our application's performance.

We will implement classical ways of optimizing a web application: cache control headers, Gzipping, an application cache, and ETags, as well as more reactive stuff, such as asynchronous method calls and WebSockets.

A production profile

In the previous chapter, we saw how to define an application properties file that will only be read while launching the application with a specific profile. We will use the same approach and create an `application-prod.properties` file in `src/main/resources`, right next to the existing `application.properties` file. This way, we will be able to configure the production environment with optimized settings.

We will put a few properties in this file to get started. In *Chapter 3, Handling Forms and Complex URL Mapping*, we deactivated the Thymeleaf cache and forced translation bundles to reload on every access.

This is great for developing but is useless and time consuming in production. So let's fix that:

```
spring.thymeleaf.cache=true
spring.messages.cache-seconds=-1
```

A cache period of `-1` means caching the bundle forever.

Now, if we launch our application with the "prod" profile, templates and bundles should be cached forever.

The properties coming from the "prod" profile will indeed overwrite the ones declared in our `application.properties` file.

Gzipping

Gzipping is a compression algorithm widely understood by browsers. Your server will serve compressed responses, which will consume a few more CPU cycles but will save bandwidth.

The client browser will then be charged for unzipping the resources and displaying them to the user.

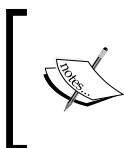
To leverage Tomcat's Gzipping abilities, simply add the following line to the `application-prod.properties` file:

```
server.tomcat.compression=on
server.tomcat.compressableMimeTypes=text/html, text/xml, text/css, text/plain, \
    application/json, application/xml, application/javascript
```

This will enable Tomcat's Gzipping compression when serving any file matching the MIME types specified in the list, and whose length is greater than 2048 bytes. You can set `server.tomcat.compression` to `force` to enforce compression or set it to a numerical value if you want to change the value for the minimal length of Gzipped assets.

If you want more control over the compression, say over the level of compression, or want to exclude user agents from compression, you can use the `GzipFilter` class in Jetty by adding the `org.eclipse.jetty:jetty-servlets` dependency to your project.

This will automatically trigger the `GzipFilterAutoConfiguration` class, which can be configured with a handful of properties prefixed by `spring.http.gzip`. Have a look at `GzipFilterProperties` to understand its level of customization.



Refer to the documentation at <http://docs.spring.io/spring-boot/docs/current/reference/html/howto-embedded-servlet-containers.html#how-to-enable-http-response-compression> for additional information.

Cache control

Cache control is a set of HTTP headers sent by the server to control how the user's browser is allowed to cache resources.

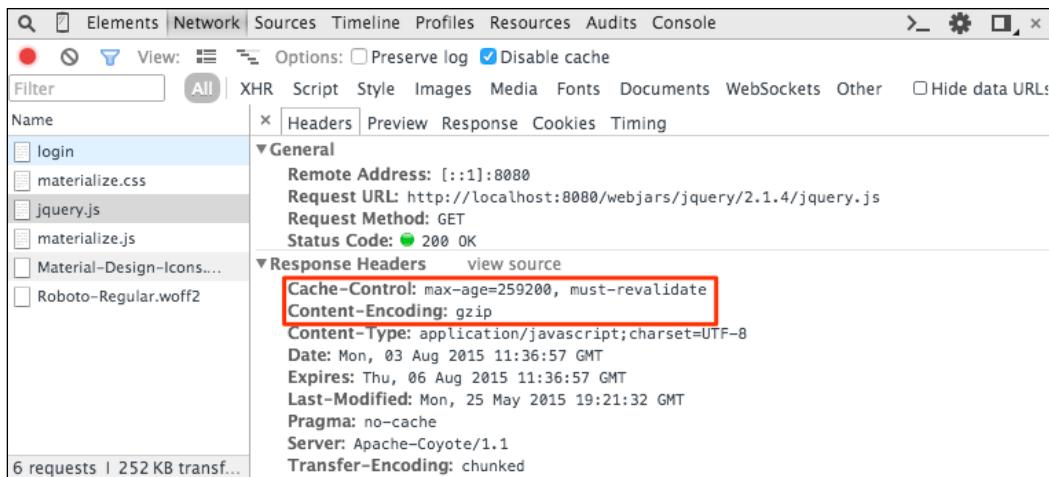
In the previous chapter, we have seen that Spring Security automatically disables caching for secured resources.

If we want to benefit from cache control, we must first disable that feature:

```
security.headers.cache=false

# Cache resources for 3 days
spring.resources.cache-period=259200
```

Now, launch the application, go to the main page, and check the Chrome developer console. You will see that our JavaScript files are Gzipped and cached, as marked in the following screenshot:



If you want more control over your cache, you could add handlers for your own resources in your configuration:

```
@Override
public void addResourceHandlers(ResourceHandlerRegistry registry) {
    // This is just an example
    registry.addResourceHandler("/img/**")
        .addResourceLocations("classpath:/static/images/")
        .setCachePeriod(12);
}
```

We could also override the Spring Security default settings. If we want to deactivate the "no cache control" policy for our API, we can change the `ApiSecurityConfiguration` class like this:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .antMatcher("/api/**")
    // This is just an example - not required in our case
        .headers().cacheControl().disable()
        .httpBasic().and()
        .csrf().disable()
        .authorizeRequests()
        .antMatchers(HttpMethod.GET).hasRole("USER")
        .antMatchers(HttpMethod.POST).hasRole("ADMIN")
        .antMatchers(HttpMethod.PUT).hasRole("ADMIN")
        .antMatchers(HttpMethod.DELETE).hasRole("ADMIN")
        .anyRequest().authenticated();
}
```

Application cache

Now that our web requests have been compressed and cached, the next step we can take to reduce server load is to put the results of costly operations in a cache. The Twitter search takes some time and will consume our application request ratio on the Twitter API. With Spring, we can easily cache the search and return the same result each time the search is called with the same parameters.

The first thing that we need to do is activate Spring caching with the `@EnableCache` annotation. We also need to create a `CacheManager` that will resolve our caches. Let's create a `CacheConfiguration` class in the `config` package:

```
package masterSpringMvc.config;

import org.springframework.cache.CacheManager;
import org.springframework.cache.annotation.EnableCaching;
import org.springframework.cache.concurrent.ConcurrentMapCache;
import org.springframework.cache.support.SimpleCacheManager;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import java.util.Arrays;
```

```
@Configuration
@EnableCaching
public class CacheConfiguration {

    @Bean
    public CacheManager cacheManager() {
        SimpleCacheManager simpleCacheManager = new
        SimpleCacheManager();
        simpleCacheManager.setCaches((Arrays.asList(
            new ConcurrentMapCache("searches")
        )));
        return simpleCacheManager;
    }
}
```

In the previous example, we use the simplest possible cache abstraction. Other implementations are also available, such as `EhCacheCacheManager` or `GuavaCacheManager`, which we will use in a moment.

Now that we have configured our cache, we can use the `@Cacheable` annotation on our methods. When we do that, Spring will automatically cache the result of the method and associate it with the current parameters for retrieval.

Spring needs to create a proxy around beans whose methods are cached. This typically means that calling a cached method inside of the same bean will not fail to use Spring's cache.

In our case, in the `SearchService` class, the part where we call the search operations, would benefit greatly from caching.

As a preliminary step, it would be good to put the code responsible for creating the `SearchParameters` class in a dedicated object called `SearchParamsBuilder`:

```
package masterSpringMvc.search;

import org.springframework.social.twitter.api.SearchParameters;

import java.util.List;
import java.util.stream.Collectors;

public class SearchParamsBuilder {

    public static SearchParameters createSearchParams(String
searchType, String taste) {
        SearchParameters.ResultType resultType =
getResultType(searchType);
    }
}
```

```
    SearchParameters searchParameters = new
    SearchParameters(taste);
    searchParameters.resultType(resultType);
    searchParameters.count(3);
    return searchParameters;
}

private static SearchParameters.ResultType getResultType(String
searchType) {
    for (SearchParameters.ResultType knownType : SearchParameters.
ResultType.values()) {
        if (knownType.name().equalsIgnoreCase(searchType)) {
            return knownType;
        }
    }
    return SearchParameters.ResultType.RECENT;
}
}
```

This will help us to create search parameters in our service.

Now we want to create a cache for our search results. We want each call to the Twitter API to be cached. Spring cache annotations rely on proxies to instrument the `@Cacheable` methods. We therefore need a new class with a method annotated with the `@Cacheable` annotation.

When you use the Spring abstraction API, you don't know about the underlying implementation of the cache. Many will require both the return type and the parameter types of the cached method to be `Serializable`.

`SearchParameters` is not `Serializable`, that's why we will pass both the search type and the keyword (both strings) in the cached method.

Since we want to put the `LightTweets` object in cache, we want to make them `Serializable`; this will ensure that they can always be written and read from any cache abstraction:

```
public class LightTweet implements Serializable {
    // the rest of the code remains unchanged
}
```

Let's create a `SearchCache` class and put it in the `search.cache` package:

```
package masterSpringMvc.search.cache;

import masterSpringMvc.search.LightTweet;
import masterSpringMvc.search.SearchParamsBuilder;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.autoconfigure.social.
TwitterProperties;
import org.springframework.cache.annotation.Cacheable;
import org.springframework.social.twitter.api.SearchParameters;
import org.springframework.social.twitter.api.Twitter;
import org.springframework.social.twitter.api.impl.TwitterTemplate;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.stream.Collectors;

@Service
public class SearchCache {
    protected final Log logger = LogFactory.getLog(getClass());
    private Twitter twitter;

    @Autowired
    public SearchCache(TwitterProperties twitterProperties) {
        this.twitter = new TwitterTemplate(twitterProperties.
getAppId(), twitterProperties.getAppSecret());
    }

    @Cacheable("searches")
    public List<LightTweet> fetch(String searchType, String keyword) {
        logger.info("Cache miss for " + keyword);
        SearchParameters searchParam = SearchParamsBuilder.
createSearchParam(searchType, keyword);
        return twitter.searchOperations()
            .search(searchParam)
            .getTweets().stream()
            .map(LightTweet::ofTweet)
            .collect(Collectors.toList());
    }
}
```

It can't really get simpler than that. We used the `@Cacheable` annotation to specify the name of the cache that will be used. Different caches may have different policies.

Note that we manually created a new `TwitterTemplate` method rather than injecting it like before. That's because we will have to access the cache from other threads a little bit later. In Spring Boot's `TwitterAutoConfiguration` class, the `Twitter` bean is bound to the request scope and is therefore not available outside of a Servlet thread.

With those two new objects, the code of our `SearchService` class simply becomes this:

```
package masterSpringMvc.search;

import masterSpringMvc.search.cache.SearchCache;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Profile;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.stream.Collectors;

@Service
@Profile("!async")
public class SearchService implements TwitterSearch {
    private SearchCache searchCache;

    @Autowired
    public SearchService(SearchCache searchCache) {
        this.searchCache = searchCache;
    }

    @Override
    public List<LightTweet> search(String searchType, List<String> keywords) {
        return keywords.stream()
            .flatMap(keyword -> searchCache.fetch(searchType,
                keyword).stream())
            .collect(Collectors.toList());
    }
}
```

Note that we annotated the service with `@Profile("!async")`. This means that we only create this bean if the profile `async` is not activated.

Later, we will create another implementation of the `TwitterSearch` class to be able to switch between the two.

Neat! Say we restart our application and try a big request such as the following:

```
http://localhost:8080/search/mixed;keywords=docker,spring,spring%20boot,spring%20mvc,groovy,grails
```

It will take a little time at first, but then our console will display the following log:

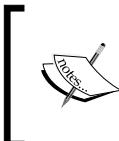
```
2015-08-03 16:04:01.958  INFO 38259 --- [nio-8080-exec-8] m.search.cache.SearchCache : Cache miss for docker
2015-08-03 16:04:02.437  INFO 38259 --- [nio-8080-exec-8] m.search.cache.SearchCache : Cache miss for spring
2015-08-03 16:04:02.728  INFO 38259 --- [nio-8080-exec-8] m.search.cache.SearchCache : Cache miss for spring boot
2015-08-03 16:04:03.098  INFO 38259 --- [nio-8080-exec-8] m.search.cache.SearchCache : Cache miss for spring mvc
2015-08-03 16:04:03.383  INFO 38259 --- [nio-8080-exec-8] m.search.cache.SearchCache : Cache miss for groovy
2015-08-03 16:04:03.967  INFO 38259 --- [nio-8080-exec-8] m.search.cache.SearchCache : Cache miss for grails
```

After that, if we hit refresh, the result will be displayed immediately and no cache miss will be seen in the console.

That's it for our cache, but there is much more to the cache API. You can annotate methods with the following:

- `@CachEvict`: This will remove an entry from the cache
- `@CachePut`: This will put the result of a method into a cache without interfering with the method itself
- `@Caching`: This regroups the caching annotation
- `@CacheConfig`: This points to different caching configurations

The `@Cacheable` annotation can also be configured to cache results on certain conditions.



For more information on Spring cache, please see the following documentation:
<http://docs.spring.io/spring/docs/current/spring-framework-reference/html/cache.html>

Cache invalidation

Currently, search results will be cached forever. Using the default simple cache manager doesn't give us a lot of options. There is one more thing that we can do to improve our application caching. Since we have Guava in our classpath, we can replace the existing cache manager in the cache configuration with the following code:

```
package masterSpringMvc.config;

import com.google.common.cache.CacheBuilder;
import org.springframework.cache.CacheManager;
import org.springframework.cache.annotation.EnableCaching;
import org.springframework.cache.guava.GuavaCacheManager;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

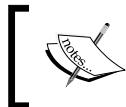
import java.util.concurrent.TimeUnit;

@Configuration
@EnableCaching
public class CacheConfiguration {

    @Bean
    public CacheManager cacheManager() {
        GuavaCacheManager cacheManager = new
GuavaCacheManager("searches");
        cacheManager
            .setCacheBuilder(
                CacheBuilder.newBuilder()
                    .softValues()
                    .expireAfterWrite(10, TimeUnit.
MINUTES)
            );
        return cacheManager;
    }
}
```

This will build a cache expiring after 10 minutes and using soft values, meaning that the entries will be cleaned up if the JVM runs low on memory.

Try to fiddle around with Guava's cache builder. You can specify a smaller time unit for your testing, and even specify different cache policies.



See the documentation at <https://code.google.com/p/guava-libraries/wiki/CachesExplained>.



Distributed cache

We already have a Redis profile. If Redis is available, we could also use it as our cache provider. It would allow us to distribute the cache across multiple servers. Let's change the RedisConfig class:

```
package masterSpringMvc.config;

import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.cache.CacheManager;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;
import org.springframework.context.annotation.Profile;
import org.springframework.data.redis.cache.RedisCacheManager;
import org.springframework.data.redis.connection.RedisConnectionFactory;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.session.data.redis.config.annotation.web.http.EnableRedisHttpSession;

import java.util.Arrays;

@Configuration
@Profile("redis")
@EnableRedisHttpSession
public class RedisConfig {

    @Bean(name = "objectRedisTemplate")
    public RedisTemplate<Object, Object> objectRedisTemplate(RedisConnectionFactory
redisConnectionFactory) {
        RedisTemplate<Object, Object> template = new
RedisTemplate<>();
        template.setConnectionFactory(redisConnectionFactory);
        return template;
    }
}
```

```
    @Primary @Bean
    public CacheManager cacheManager(@Qualifier("objectRedisTemplate"))
RedisTemplate template) {
    RedisCacheManager cacheManager = new
RedisCacheManager(template);
    cacheManager.setCacheNames(Arrays.asList("searches"));
    cacheManager.setDefaultExpiration(36_000);
    return cacheManager;
}
}
```

With this configuration, if we run our application with the "Redis" profile, the Redis cache manager will be used instead of the one defined in the CacheConfig class since it is annotated with `@Primary`.

This will allow the cache to be distributed in case we want to scale on more than one server. The Redis template is used to serialize the cache return values and parameters, and will require objects to be `Serializable`.

Async methods

There is still a bottleneck in our application; when a user searches ten keywords, each search will be executed sequentially. We could easily improve the speed of our application by using different threads and launching all the searches at the same time.

To enable Spring's asynchronous capabilities, one must use the `@EnableAsync` annotation. This will transparently execute any method annotated with `@Async` using a `java.util.concurrent.Executor`.

It is possible to customize the default executor used by implementing the `AsyncConfigurer` interface. Let's create a new configuration class called `AsyncConfig` in the `config` package:

```
package masterSpringMvc.config;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.aop.interceptor.
AsyncUncaughtExceptionHandler;
import org.springframework.context.annotation.Configuration;
import org.springframework.scheduling.annotation.AsyncConfigurer;
import org.springframework.scheduling.annotation.EnableAsync;
```

```
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;

@Configuration
@EnableAsync
public class AsyncConfiguration implements AsyncConfigurer {

    protected final Log logger = LoggerFactory.getLog(getClass());

    @Override
    public Executor getAsyncExecutor() {
        return Executors.newFixedThreadPool(10);
    }

    @Override
    public AsyncUncaughtExceptionHandler
    getAsyncUncaughtExceptionHandler() {
        return (ex, method, params) -> logger.error("Uncaught async
error", ex);
    }
}
```

With this configuration, we ensure that no more than 10 threads will be allocated to handle our asynchronous tasks in the whole application. This is very important in a web application where each client has a dedicated thread. The more threads you use and the longer they block, the fewer client requests you can process.

Let's annotate our search method and make it asynchronous. We will need to make it return a subtype of `Future`, a java concurrent class that represents an asynchronous result.

We will create a new implementation of the `TwitterSearch` class that will query the search API in different threads. The implementation is a bit tricky so I'll break it down into small parts.

First, we need to annotate the method that will query the API with the `@Async` annotation to tell Spring to schedule the task using our executor. Again, Spring will use proxy to do its magic so this method has to be in a different class to the service calling it. It would also be nice if this component could use our cache. That would lead us to create this component:

```
@Component
private static class AsyncSearch {
```

```
protected final Log logger = LogFactory.getLog(getClass());
private SearchCache searchCache;

@Autowired
public AsyncSearch(SearchCache searchCache) {
    this.searchCache = searchCache;
}

@Async
public ListenableFuture<List<LightTweet>> asyncFetch(String
searchType, String keyword) {
    logger.info(Thread.currentThread().getName() + " - Searching
for " + keyword);
    return new AsyncResult<>(searchCache.fetch(searchType,
keyword));
}
```

Don't create this class yet. Let's see what our service needs first.

The `ListenableFuture` abstraction allows us to add callbacks after the completion of the future, either in the case of correct results or if an exception occurs.

The algorithm to wait for a bunch of asynchronous tasks would look like this:

```
@Override
public List<LightTweet> search(String searchType, List<String>
keywords) {
    CountDownLatch latch = new CountDownLatch(keywords.size());
    List<LightTweet> allTweets = Collections.synchronizedList(new
ArrayList<>());
    keywords
        .stream()
        .forEach(keyword -> asyncFetch(latch, allTweets,
searchType, keyword));

    await(latch);
    return allTweets;
}
```

If you don't know the `CountDownLatch` method, it is just a simple blocking counter.

The `await()` method will wait until the latch reaches 0 to unlock the thread.

The `asyncFetch` method, shown in the preceding code, will attach a callback to each of our `asynFetch` methods. The callback will add the results to the `allTweets` list and decrement the latch. Once each callback has been called, the method will return all the tweets.

Got it? Here is the final code:

```
package masterSpringMvc.search;

import masterSpringMvc.search.cache.SearchCache;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Profile;
import org.springframework.scheduling.annotation.Async;
import org.springframework.scheduling.annotation.AsyncResult;
import org.springframework.social.twitter.api.SearchParameters;
import org.springframework.stereotype.Component;
import org.springframework.stereotype.Service;
import org.springframework.util.concurrent.ListenableFuture;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.concurrent.CountDownLatch;

@Service
@Profile("async")
public class ParallelSearchService implements TwitterSearch {
    private final AsyncSearch asyncSearch;

    @Autowired
    public ParallelSearchService(AsyncSearch asyncSearch) {
        this.asyncSearch = asyncSearch;
    }

    @Override
    public List<LightTweet> search(String searchType, List<String>
keywords) {
        CountDownLatch latch = new CountDownLatch(keywords.size());
        List<LightTweet> allTweets = Collections.synchronizedList(new
ArrayList<>());
        latch.countDown();
        for (String keyword : keywords) {
            asyncSearch.fetch(keyword, new AsyncResultCallback<List<LightTweet>>() {
                @Override
                public void onResult(List<LightTweet> result) {
                    allTweets.addAll(result);
                    latch.countDown();
                }
            });
        }
        latch.await();
        return allTweets;
    }
}
```

```
    keywords
        .stream()
        .forEach(keyword -> asyncFetch(latch, allTweets,
    searchType, keyword));

        await(latch);
        return allTweets;
    }

    private void asyncFetch(CountDownLatch latch, List<LightTweet>
allTweets, String searchType, String keyword) {
    asyncSearch.asyncFetch(searchType, keyword)
        .addCallback(
            tweets -> onSuccess(allTweets, latch, tweets),
            ex -> onError(latch, ex));
}

private void await(CountDownLatch latch) {
    try {
        latch.await();
    } catch (InterruptedException e) {
        throw new IllegalStateException(e);
    }
}

private static void onSuccess(List<LightTweet> results,
CountDownLatch latch, List<LightTweet> tweets) {
    results.addAll(tweets);
    latch.countDown();
}

private static void onError(CountDownLatch latch, Throwable ex) {
    ex.printStackTrace();
    latch.countDown();
}

@Component
private static class AsyncSearch {
    protected final Log logger = LogFactory.getLog(getClass());
    private SearchCache searchCache;

    @Autowired
    public AsyncSearch(SearchCache searchCache) {
        this.searchCache = searchCache;
    }
}
```

```
    }

    @Async
    public ListenableFuture<List<LightTweet>> asyncFetch(String
searchType, String keyword) {
        logger.info(Thread.currentThread().getName() + " - "
Searching for " + keyword);
        return new AsyncResult<>(searchCache.fetch(searchType,
keyword));
    }
}
```

Now, to use this implementation, we need to run the application with the `async` profile.

We can run it with multiple profiles active at the same time by separating them with commas, as follows:

```
--spring.profiles.active=redis,async
```

If we launch a search on multiple terms, we can see something like this:

```
pool-1-thread-3 - Searching groovy
pool-1-thread-1 - Searching spring
pool-1-thread-2 - Searching java
```

This shows that the different searches are done in parallel.

Java 8 actually introduced a new type called `CompletableFuture`, which is a much better API to manipulate futures. The main problem with completable futures is that no executor can work with them without a bit of code. This is outside of the scope of the article, but you can check my blog for an article on the subject: <http://geowarin.github.io/spring/2015/06/12/completable-futures-with-spring-async.html>.



Disclaimer

The following sections contains a lot of JavaScript. Obviously, I think you should have a look at the code, especially if JavaScript is not your favorite language. It is time to learn it. That being said, even if WebSocket is insanely cool, it is not a requirement. You can safely skip ahead to the last chapter and deploy your application right now.

ETags

Our Twitter results are neatly cached, so a user refreshing the result page will not trigger an additional search on the Twitter API. However, the response will be sent to this user multiple times even if the results do not change, which will waste bandwidth.

An ETag is a hash of the data of a web response and is sent as a header. The client can memorize the ETag of a resource and send the last known version to the server with the `If-None-Match` header. This allows the server to answer `304 Not Modified` if the request does not change in the meantime.

Spring has a special Servlet filter, called `ShallowEtagHeaderFilter`, to handle ETags. Simply add it as a bean in the `MasterSpringMvc4Application` configuration class:

```
@Bean
public Filter etagFilter() {
    return new ShallowEtagHeaderFilter();
}
```

This will automatically generate ETags for your responses as long as the response has no cache control headers.

Now if we interrogate our RESTful API, we can see that an ETag is sent along with the server response:

```
> http GET 'http://localhost:8080/api/search/mixed;keywords=spring' -a
admin:admin

HTTP/1.1 200 OK
Content-Length: 1276
Content-Type: application/json; charset=UTF-8
Date: Mon, 01 Jun 2015 11:29:51 GMT
ETag: "00a66d6dd835b6c7c60638eab976c4dd7"
Server: Apache-Coyote/1.1
Set-Cookie: JSESSIONID=662848E4F927EE9A1BA2006686ECFE4C; Path=/; HttpOnly
```

Now if we request the same resource one more time, specifying the last ETag that we know of in the `If-None-Match` headers, the server will automatically respond with a `304 Not Modified` status:

```
> http GET 'http://localhost:8080/api/search/mixed;keywords=spring' If-
None-Match:'"00a66d6dd835b6c7c60638eab976c4dd7"' -a admin:admin

HTTP/1.1 304 Not Modified
```

```
Date: Mon, 01 Jun 2015 11:34:21 GMT
ETag: "00a66d6dd835b6c7c60638eab976c4dd7"
Server: Apache-Coyote/1.1
Set-Cookie: JSESSIONID=CA956010CF268056C241B0674C6C5AB2; Path=/; HttpOnly
```



Due to the parallel nature of our search, the tweets fetched for different keywords might arrive in different orders, which will make the ETag change. If you want this technique to work for multiple searches, please consider ordering your search results before sending them to the client.

If we want to take advantage of that, we obviously need to rewrite our client code to handle them. We will see a simple solution to do that with jQuery, using the local storage of the browser to save the latest query of the user.

First, remove the `tweets` variable from our model; we won't do the search from the server anymore. You will have to modify a test or two to reflect this change.

Before going further, let's add lodash to our JavaScript libraries. If you don't know lodash, let's say it is the Apache Utils of JavaScript. You can add it to your project dependencies like so:

```
compile 'org.webjars.bower:lodash:3.9.3'
```

Add it to the `default.html` layout, just under the materialize's JavaScript:

```
<script src="/webjars/lodash/3.9.3/lodash.js"></script>
```

We will modify the `resultPage.html` file and leave the part where the tweets should appear empty:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      layout:decorator="layout/default">
<head lang="en">
    <title>Hello twitter</title>
</head>
<body>
    <div class="row" layout:fragment="content">
        <h2 class="indigo-text center" th:text="|Tweet results for
${search}|">Tweets</h2>
```

```
<ul id="tweets" class="collection">
</ul>
</div>
</body>
</html>
```

Then, we will add a script element at the bottom of the page, just before closing the body:

```
<script layout:fragment="script" th:inline="javascript">
/*<![CDATA[*/
    var baseUrl = /*[@{/api/search}]]*/ "/";
    var currentLocation = window.location.href;
    var search = currentLocation.substr(currentLocation.
lastIndexOf('/'));
    var url = baseUrl + search;
/*]]&gt;*/
&lt;/script&gt;</pre>
```

The preceding script will just be in charge of constructing the URL for our request. We will use it by issuing a simple jQuery AJAX call:

```
$.ajax({
    url: url,
    type: "GET",
    beforeSend: setEtag,
    success: onResponse
});
```

We will use the `beforeSend` callback to have a chance to modify the request headers just before the call is made:

```
function getLastQuery() {
    return JSON.parse(localStorage.getItem('lastQuery')) || {};
}

function storeQuery(query) {
    localStorage.setItem('lastQuery', JSON.stringify(query));
}

function setEtag(xhr) {
    xhr.setRequestHeader('If-None-Match', getLastQuery().etag)
}
```

As you can see, we can easily read and write from local storage. The gotcha here is that local storage only works with strings so we have to parse and serialize the query object to JSON.

We can handle the response by retrieving the content from local storage if the HTTP status is 304 Not Modified:

```
function onResponse(tweets, status, xhr) {
  if (xhr.status == 304) {
    console.log('Response has not changed');
    tweets = getLastQuery().tweets
  }

  var etag = xhr.getResponseHeader('Etag');
  storeQuery({tweets: tweets, etag: etag});

  displayTweets(tweets);
}

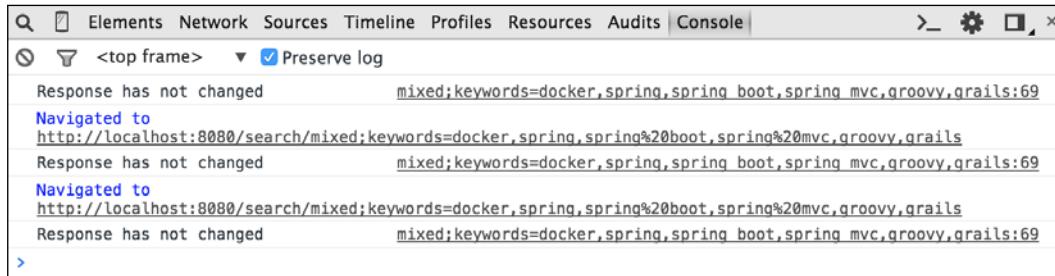
function displayTweets(tweets) {
  $('#tweets').empty();
  $.each(tweets, function (index, tweet) {
    addTweet(tweet);
  })
}
```

For the `addTweet` function that you will see next, I'm using lodash, a very useful JavaScript utility library, to generate templates. The function to add tweets to the page can be written as follows:

```
function addTweet(tweet) {
  var template = _.template('<li class="collection-item avatar">' +
    '' +
    '<span class="title">${tweet.user}</span>' +
    '<p>${tweet.text}</p>' +
    '</li>');
  $('#tweets').append(template({tweet: tweet}));
}
```

That was a lot of JavaScript! It would make more sense to generalize this pattern in a Single Page Application using a library such as Backbone.js. Hopefully, though, this will serve as a simple example of how to implement ETags in your application.

If you try to refresh the search page multiple times, you will see that the contents do not change and will be displayed immediately:



The screenshot shows the Network tab of the Chrome DevTools. It lists several requests for the URL `http://localhost:8080/search/mixed;keywords=docker,spring,spring boot,spring mvc,groovy,grails`. Each request is labeled 'Response has not changed' and has a timestamp. The requests are timestamped at approximately 10:00, 10:01, 10:02, and 10:03. The status bar at the bottom of the DevTools shows the message 'mixed;keywords=docker,spring,spring boot,spring mvc,groovy,grails:69'.

There are other uses for ETags, such as optimistic locking for transactions (it lets you know on which version of an object the client is supposed to be working on at any time). It is also extra work on the server side to hash the data before sending it across, but it will save bandwidth.

WebSockets

Another kind of optimization we can think about is sending the data to the client as it becomes available to the server. Since we fetch results of the search in multiple threads, the data will come in multiple chunks. We could send them bit by bit instead of waiting for all the results.

Spring has excellent support for WebSockets, which is a protocol that allows clients to maintain a long-running connection to the server. Data can be pushed in web sockets on both ends of the connection and consumers will get the data in real-time.

We will use a JavaScript library called SockJS to ensure compatibility with all browsers. Sockjs will transparently fall back on another strategy if our users have an outdated browser.

We will also use StompJS to connect to our message broker.

Add the following library to your build:

```
compile 'org.springframework.boot:spring-boot-starter-websocket'  
compile 'org.springframework:spring-messaging'  
  
compile 'org.webjars:sockjs-client:1.0.0'  
compile 'org.webjars:stomp-websocket:2.3.3'
```

Add the WebJars to our default Thymeleaf template:

```
<script src="/webjars/sockjs-client/1.0.0/sockjs.js"></script>
<script src="/webjars/stomp-websocket/2.3.3/stomp.js"></script>
```

To configure WebSockets in our application, we need to add a bit of configuration as well:

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfiguration extends
AbstractWebSocketMessageBrokerConfigurer {

    @Override
    public void configureMessageBroker(MessageBrokerRegistry config) {
        config.enableSimpleBroker("/topic");
        config.setApplicationDestinationPrefixes("/ws");
    }

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry)
    {
        registry.addEndpoint("/twitterSearch").withSockJS();
    }

}
```

This will configure the different channels available in our application. SockJS clients will connect to the `twitterSearch` endpoint and will push data to the server on `/ws/` channel and be able to listen to `/topic/` for changes.

This will allow us to inject a `SimpMessagingTemplate` in a new controller to push data to the client in the `/topic/searchResult` channel, as follows:

```
@Controller
public class SearchSocketController {
    private CachedSearchService searchService;
    private SimpMessagingTemplate webSocket;

    @Autowired
    public SearchSocketController(CachedSearchService searchService,
        SimpMessagingTemplate webSocket) {
        this.searchService = searchService;
        this.webSocket = webSocket;
    }
}
```

```
    @MessageMapping("/search")
    public void search(@RequestParam List<String> keywords) throws
    Exception {
        Consumer<List<LightTweet>> callback = tweet -> webSocket.
        convertAndSend("/topic/searchResults", tweet);
        twitterSearch(SearchParameters.ResultType.POPULAR, keywords,
        callback);
    }

    public void twitterSearch(SearchParameters.ResultType resultType,
    List<String> keywords, Consumer<List<LightTweet>> callback) {
        keywords.stream()
            .forEach(keyword -> {
                searchService.search(resultType, keyword)
                    .addCallback(callback::accept,
                    Throwable::printStackTrace);
            });
    }
}
```

In our `resultPage`, the JavaScript code is really simple:

```
var currentLocation = window.location.href;
var search = currentLocation.substr(currentLocation.lastIndexOf('=') +
1);

function connect() {
    var socket = new SockJS('/hello');
    stompClient = Stomp.over(socket);
    // stompClient.debug = null;
    stompClient.connect({}, function (frame) {
        console.log('Connected: ' + frame);

        stompClient.subscribe('/topic/searchResults', function (result)
{
            displayTweets(JSON.parse(result.body));
        });

        stompClient.send("/app/search", {}, JSON.stringify(search.
split(',')));
    });
}
```

The `displayTweets` function remains essentially the same as before:

```
function displayTweets(tweets) {
    $.each(tweets, function (index, tweet) {
        addTweet(tweet);
    })
}

function addTweet(tweet) {
    var template = _.template('<li class="collection-item avatar">' +
        '' +
        '<span class="title">${tweet.userName}</span>' +
        '<p>${tweet.text}</p>' +
        '</li>');
    $('#tweets').append(template({tweet: tweet}));
}
```

Here you go! The client will now receive the results of all the searches in the application-- live!

Before pushing this to production, it will require a little bit more work. Here are some ideas:

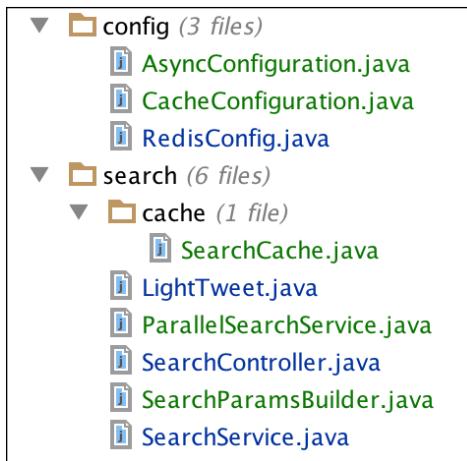
- Create subchannels for clients to privately listen to changes
- Close the channel when a client is done using it
- Add CSS transitions to the new tweets so the user can feel that it's real-time
- Use a real broker, such as RabbitMQ, to allow the backend to scale with connections

There is much more to WebSocket than just this simple example. Don't forget to have a look at the documentation at <http://docs.spring.io/spring/docs/current/spring-framework-reference/html/websocket.html> for more information.

The check point

In this chapter, we created two new configurations: `AsyncConfiguration`, which will allow us to use the `@Async` annotation to submit tasks to an executor, and `CacheConfiguration`, which will create a `CacheManager` interface and allow us to use the `@Cacheable` annotation. Since we can use Redis as a cache manager, we also amended the `RedisConfig` class.

We created a `SearchCache` class, which contained a cache of tweets, and we now have two `TwitterSearch` implementations to choose from: good old `SearchService`, which will fetch each result synchronously, and `ParallelSearchService`, which will issue each query in a different thread:



Summary

In this chapter, we have seen two different philosophies relating to performance improvement. At the beginning, we tried to reduce the bandwidth used by our clients by caching data and using as few connections to our server as possible.

In the second part, though, we began to do something more advanced by allowing searches to be run in parallel and each client to remain in sync with a persistent connection to the server through web sockets. This will allow clients to receive updates in real time, and our application will feel more reactive but consume more threads.

I strongly encourage you to polish the result before we move on to the next chapter and deploy our application for good!

9

Deploying Your Web Application to the Cloud

In this chapter, we'll take a tour of the different cloud providers, understand the challenges and benefits of a distributed architecture, and see how to deploy your web application to Pivotal Web Services and to Heroku.

Choosing your host

There are many forms of cloud hosting. For developers, the choice will be mainly between a Platform as a Service (PaaS) and an Infrastructure as a Service (IaaS).

Using the latest, you will often have a bare metal machine that you can manage and on which you can install all the services required by your application.

If we leave aside technologies such as Docker (which is absolutely amazing, you should absolutely give it a try), this is really similar to traditional hosting where your operation team will have to set up and maintain an environment in which the application can run.

On the other hand, PaaS makes it easy to deploy your application as you develop it with a simple push-to-deploy workflow.

The most well known providers are:

- Cloud Foundry backed by Pivotal
- OpenShift by Red Hat
- Heroku acquired by Salesforce in 2010

Each of these three providers come with different pros and cons. I will try to give you an overview of these.

Cloud Foundry

Backed by Pivotal, the company behind Spring, Pivotal Web Service runs on Cloud Foundry, an open source PaaS maintained by a foundation, and comes with an interesting package.

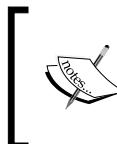
They offer a 60 day free trial and their pricing is a function of the memory allocated for your instances and the number of instances you own.

Their prices range from \$2.70 per month for the smallest (128 Mb) instance to \$43.20 per month for the 2 GB instance.

If you want to give it a try, no credit card is required for the free trial. They have a market place to easily install services, such as Redis or Postgre SQL, with rather limited free options. They have a good command-line utility to manage your application from your console. You can either use buildpacks or push a JAR directly for deployment.



Build packs will try to guess the stack that you are using and build your application in the most standard way (`mvn package` for Maven, `./gradlew stage` for Gradle, and so on).



Refer to the tutorial available at the following URL to deploy your application to Cloud Foundry:
<http://docs.cloudfoundry.org/buildpacks/java/gsg-spring.html>

OpenShift

OpenShift is maintained by Red Hat and powered by OpenShift Origin, an open source facility running Docker containers on top of Google's Kubernetes.

It is priced well and offers a lot of freedom, as it is both a PaaS and an IaaS. Its pricing is based on gears, containers running an application, or a service such as Jenkins, or a database.

OpenShift has a free plan offering three small gears. Your application must be idle for 24 hours per month unless you enter your billing information.

Additional or bigger gears are billed at approximately \$15 a month for the smallest, and \$72 for the biggest.

To deploy a Spring Boot application on OpenShift, you will have to use the Do It Yourself cartridge. It is a bit more work than other buildpack-based PaaS but it is also easier to configure.

Take a look at the blog post for a Spring Boot tutorial with OpenShift, which is available at <http://blog.codeleak.pl/2015/02/openshift-diy-build-spring-boot.html>.

Heroku

Heroku is a well known PaaS with extensive documentation and a code-centric approach based on build packs. It can connect to a lot of services called add-ons, but using them requires your billing information.

It is really interesting for a free project and is very fast to get started with. The downside is that it directly costs more than \$25 per month if you want to scale up. Free instances will go into the sleep mode after 30 minutes of inactivity, which means free Heroku apps will always take as much as 30 seconds to load.

Heroku has a great administration dashboard and command-line tools. For this chapter, I chose Heroku because it is very straightforward. The concepts you will grasp here are applicable to most PaaS.

You can follow most of the chapter and deploy your application without providing your credit card information as long as you do not use the Redis add-on. You won't be charged if you select the free plan.

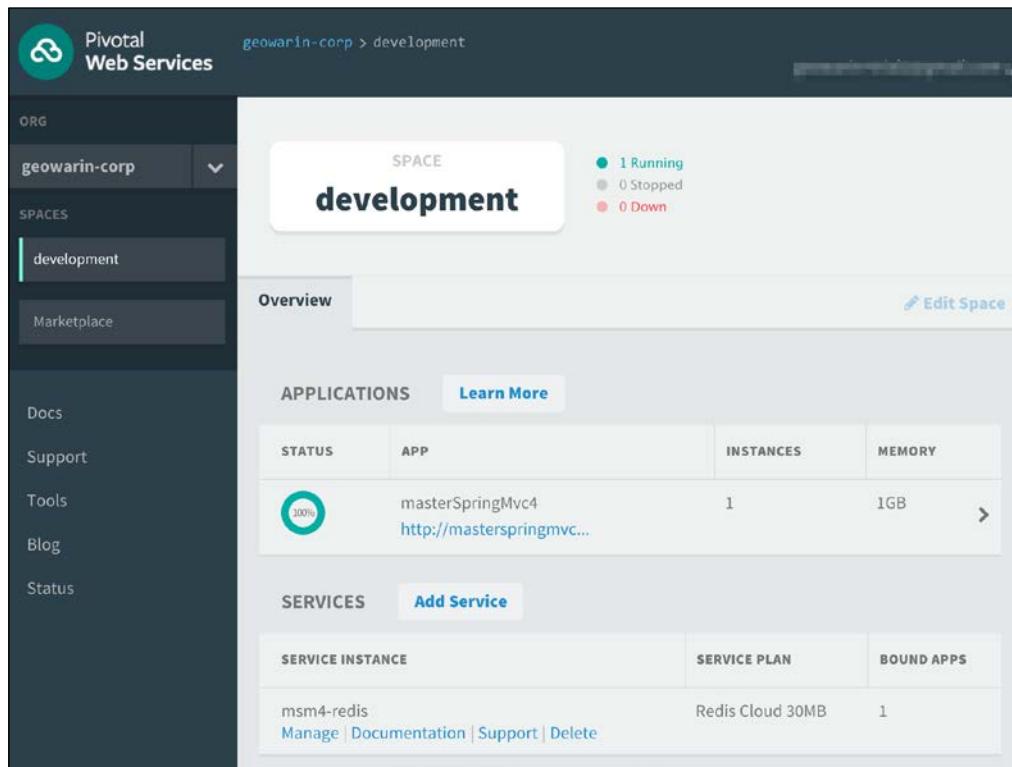
Deploying your web application to Pivotal Web Services

Follow this section if you want to deploy your application to Pivotal Web Services (PWS).

Installing the Cloud Foundry CLI tools

The first thing we need to do to create a Cloud Foundry application is to set up an account on PWS. This is documented at <http://docs.run.pivotal.io/starting/>.

You will be asked to create an organization and each new organization will have a default space (development) created within the organization. As shown in the following screenshot:



The screenshot shows the Pivotal Web Services developer console interface. The left sidebar has a 'Tools' link under the 'Support' section. The main area shows the 'development' space with one application running: 'masterSpringMvc4' (status 100%, 1 instance, 1GB memory). It also shows a service instance 'msm4-redis' bound to the application. The 'Overview' tab is selected.

On the left-hand side navigation bar, you will see a link to **Tools** from which you download the CLI. It is also available from the developer console. Select the appropriate package for your operating system:



The screenshot shows the 'Tools' section of the Pivotal Web Services developer console. It displays download links for the CLI: Windows 64 bit, Mac OSX 64 bit, Linux 64 bit (.deb), Linux 64 bit (.rpm), Windows 32 bit, Linux 32 bit (.deb), and Linux 32 bit (.rpm).

Assembling the application

Our application simply needs to be assembled for deployment.

The good thing with PWS is that you don't have to push your sources to deploy. You can generate the JAR, push it, and everything will be autodetected.

We can package this for deployment with the following command:

```
./gradlew assemble
```

This will create a jar file in the build/libs directory. At this point, you can execute the following command. The following command targets your deployment to your space within PWS (run.pivotal.io):

```
$ cf login -a api.run.pivotal.io -u <account email> -p <password> -o <organization> -s development
```

```
API endpoint: api.run.pivotal.io
Authenticating...
OK
```

```
Targeted org <account org>
```

```
Targeted space development
```

```
API endpoint: https://api.run.pivotal.io (API version: 2.33.0)
User:          <account email>
Org:          <account organization>
Space:        <account space>
```

Once you have successfully logged in, you can push your jar with the following command. You will need to come up with an available name:

```
$ cf push your-app-name -p build/libs/masterSpringMvc-0.0.1-SNAPSHOT.jar

Creating app msmvc4 in org Northwest / space development as wlund@pivotal.io...
OK
Creating route msmvc4.cfapps.io...
OK
Binding msmvc4.cfapps.io to msmvc4...
OK
Uploading msmvc4...
Uploading app files from: build/libs/masterSpringMvc-0.0.1-SNAPSHOT.jar
Uploading 690.8K, 108 files
Done uploading
OK
Starting app msmvc4 in org <Organization> / space development as <account email>
----> Downloaded app package (15M)
----> Java Buildpack Version: v3.1 | https://github.com/cloudfoundry/java-buildpack.git#7a538fb
----> Downloading Open Jdk JRE 1.8.0_51 from https://download.run.pivotal.io/openjdk/trusty/x86_64/openjdk-1.8.0_51.tar.gz (1.5s)
      Expanding Open Jdk JRE to .java-buildpack/open_jdk_jre (1.4s)
----> Downloading Open JDK Like Memory Calculator 1.1.1_RELEASE from https://download.run.pivotal.io/memory-calculator/trusty/x86_64/memory-calculator-1.1.1_RELEASE (0.1s)
      Memory Settings: -Xmx768M -Xms768M -XX:MaxMetaspaceSize=104857K -XX:MetaspaceSize=104857K -Xss1M
----> Downloading Spring Auto Reconfiguration 1.7.0_RELEASE from https://download.run.pivotal.io/auto-reconfiguration/auto-reconfiguration-1.7.0_RELEASE.jar (0.0s)
----> Uploading droplet (59M)
```

```
0 of 1 instances running, 1 starting
1 of 1 instances running

App started
OK

App msmvc4 was started using this command `CALCULATED_MEMORY=$($PWD/
java-buildpack/open_jdk_jre/bin/java-buildpack-memory-calculator-1.1.1_
RELEASE -memorySizes=metaspace:64m.. -memoryWeights=heap:75,metaspace:10
,stack:5,native:10 -totMemory=$MEMORY_LIMIT) && SERVER_PORT=$PORT $PWD/
java-buildpack/open_jdk_jre/bin/java -cp $PWD/:$PWD/.java-buildpack/
spring_auto_reconfiguration/spring_auto_reconfiguration-1.7.0_RELEASE.
jar -Djava.io.tmpdir=$TMPDIR -XX:OnOutOfMemoryError=$PWD/.java-buildpack/
open_jdk_jre/bin/killjava.sh $CALCULATED_MEMORY org.springframework.boot.
loader.JarLauncher`

Showing health and status for app msmvc4 in org <Organization> / space
development as <Account Email>
OK

requested state: started
instances: 1/1
usage: 1G x 1 instances
urls: msmvc4.cfapps.io
last uploaded: Tue Jul 28 22:04:08 UTC 2015
stack: cflinuxfs2
buildpack: java-buildpack=v3.1-https://github.com/cloudfoundry/java-
buildpack.git#7a538fb java-main open-jdk-like-jre=1.8.0_51 open-jdk-like-
memory-calculator=1.1.1_RELEASE spring-auto-reconfiguration=1.7.0_RELEASE

      state      since            cpu    memory      disk
details
#0  running  2015-07-28 03:05:04 PM  0.0%  450.9M of 1G  137M of 1G
```

There is a lot that the platform is performing on your behalf. It provisions a container and detects which buildpack is needed, in this case, Java.

It then installs the required JDK and uploads the application we pointed it to. It creates a route to the application, which it reports to us, and then launches the application for us.

Now you can view the application on the developer console:

SPACE **development**

3 Running
1 Stopped
0 Down

Overview [Edit Space](#)

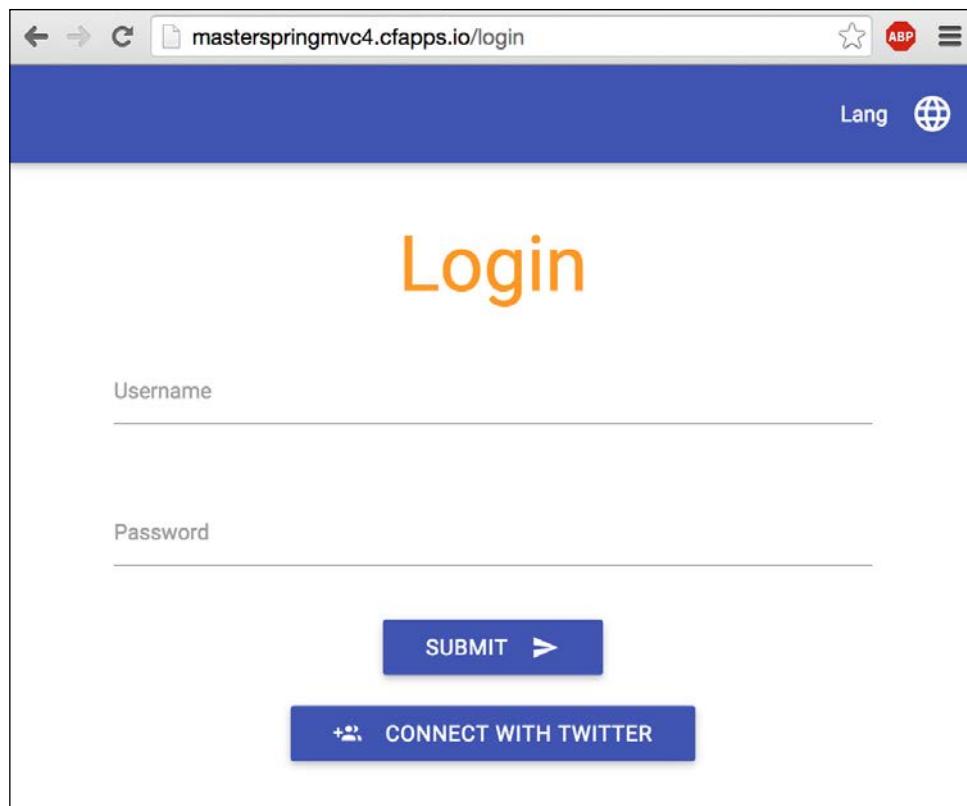
APPLICATIONS [Learn More](#)

STATUS	APP	INSTANCES	MEMORY	
100%	achilles http://achilles.cfapps.io	1	1GB	>
100%	box45 http://box45.cfapps.io	3	1GB	>
100%	msmvc4 http://msmvc4.cfapps.io	1	1GB	>
100%	mvc4-forms http://mvc4-forms.cfap...	1	1GB	>

SERVICES [Add Service](#)

SERVICE INSTANCE	SERVICE PLAN	BOUND APPS
mccmonitor Delete	User Provided	0

On selecting the highlighted route, the application will be available for use. Visit <http://msmvc4.cfapps.io>, then you will see the following screenshot:



Bravo!

The only thing that will not work yet is the file upload. However, we will fix that in a minute.

Activating Redis

In your application services, you can choose between many services. One of them is Redis Cloud, which has a free plan with 30 MB of storage. Go ahead and select this plan.

In the form, choose whatever name you fancy and bind the service to your application. By default, Cloud Foundry will inject some properties in relation to the service in your environment:

- `cloud.services.redis.connection.host`
- `cloud.services.redis.connection.port`
- `cloud.services.redis.connection.password`
- `cloud.services.redis.connection.uri`

These properties will always follow the same convention, so it will be easy to keep track of your services as you add more.

By default, Cloud Foundry launches Spring applications and activates the Cloud profile.

We can take advantage of this and create an `application-cloud.properties` file in `src/main/resources`, which will be used when our application is running on PWS:

```
spring.profiles.active=prod,redis

spring.redis.host=${cloud.services.redis.connection.host}
spring.redis.port=${cloud.services.redis.connection.port}
spring.redis.password=${cloud.services.redis.connection.password}

upload.pictures.uploadPath=file:/tmp
```

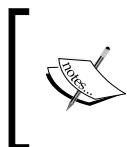
This will bind our Redis instance to our application and activate two additional profiles: `prod` and `redis`.

We also changed the path where the uploaded pictures will land. Note that using the file system on the cloud obeys different rules. Refer to the following link for more details:

<http://docs.run.pivotal.io/devguide/deploy-apps/prepare-to-deploy.html#filesystem>

The last thing we need to do is deactivate one Spring Session feature that will not be available on our hosted instance:

```
@Bean
@Profile({"cloud", "heroku"})
public static ConfigureRedisAction configureRedisAction() {
    return ConfigureRedisAction.NO_OP;
}
```



For more information, visit <http://docs.spring.io/spring-session/docs/current/reference/html5/#api-redisoperationsessionrepository-sessiondestroyedevent>.

You will see that this configuration will also be applied on Heroku.

That's it. You can reassemble your web application and push it again. Now, your sessions and application cache will be stored on Redis!

You may want to explore the marketplace for other available features such as binding to data or messaging services, scaling the application, and managing the health of the applications that are beyond the scope of this introduction.

Have fun and enjoy the productivity the platform provides!

Deploying your web application on Heroku

In this section, we will deploy your application on Heroku for free. We will even use the free Redis instance available to store our session and cache.

Installing the tools

The first thing we need to do to create a Heroku application is to download the command-line tools available at <https://toolbelt.heroku.com>.

On Mac, you can also install it with brew command:

```
> brew install heroku-toolbelt
```

Create an account on Heroku and use `heroku login` to link the toolbelt to your account:

```
> heroku login
Enter your Heroku credentials.
Email: geowarin@mail.com
Password (typing will be hidden):
Authentication successful.
```

Then, go to your application root and type `heroku create appName --region eu`. Replace `appName` with a name of your choice. If you don't provide a name, it will be generated automatically:

```
> heroku create appname --region eu
Creating appname... done, region is eu
https://appname.herokuapp.com/ | https://git.heroku.com/appname.git
Git remote heroku added
```

If you have already created an application with the UI, then go to your application root and simply add the remote `heroku git:remote -a yourapp`.

What these commands do is add a Git remote called `heroku` to our Git repository. The process of deploying on Heroku is just pushing one of your branches to Heroku. The Git hooks installed on the remote will take care of the rest.

If you type `git remote -v` command, you should see the heroku version:

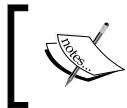
```
> git remote -v
heroku    https://git.heroku.com/appname.git (fetch)
heroku    https://git.heroku.com/appname.git (push)
origin    https://github.com/Mastering-Spring-MVC-4/mastering-spring-
          mvc4-code.git (fetch)
origin    https://github.com/Mastering-Spring-MVC-4/mastering-spring-
          mvc4-code.git (push)
```

Setting up the application

We need two ingredients to run a Gradle application with Heroku: a task in our build file called `stage` and a tiny file that contains the command used to run our application, called `ProcFile`.

Gradle

The Gradle build pack will automatically try to run the `./gradlew stage` command on the root of your application.



You can get more information on the Gradle build pack at <https://github.com/heroku/heroku-buildpack-gradle>.



We do not have a "stage" task yet. Add the following code to your `build.gradle` file:

```
task stage(type: Copy, dependsOn: [clean, build]) {
    from jar.archivePath
    into project.rootDir
    rename {
        'app.jar'
    }
}
stage.mustRunAfter(clean)

clean << {
    project.file('app.jar').delete()
}
```

This will define a task called `stage`, which will copy the jar generated by Spring Boot at the root of the application and call it `app.jar`.

The jar be much easier to find this way. The `stage` task depends on the `clean` task and the `build` task, which means that both of them will be executed before the `stage` task starts.

By default, Gradle will try to optimize the task dependency graph. So, we must provide a hint and force the `clean` task to be run before `stage`.

Finally, we add a new instruction to the already existing `clean` task, which is to delete the generated `app.jar` file.

Now, if you run `./gradlew stage`, it should run the tests and put the packaged app at the root of the project.

Procfile

When Heroku detects a Gradle application, it will automatically run a container with Java 8 installed. So, we have very little configuration to take care of.

We will need a file containing the shell command used to run our application. Create a file named `Procfile` at the root of your application:

```
web: java -Dserver.port=$PORT -Dspring.profiles.active=heroku,prod  
-jar app.jar
```

There are several things to note here. First, we declare our application as a web application. We also redefine the port on which our application will run using an environment variable. This is very important as your app will cohabit with many others and only one port will be allocated to each one.

Finally, you can see that our application will run using two profiles. The first is the `prod` profile, which we created in the previous chapter, to optimize the performance, and a new `heroku` profile that we will create in a moment.

A Heroku profile

We do not want to put sensible information, such as our Twitter app keys, into source control. So, we have to create some properties that will read those from the application environment:

```
spring.social.twitter.appId=${twitterAppId}  
spring.social.twitter.appSecret=${twitterAppSecret}
```

For this to work, you have to configure the two environment variables, which we discussed earlier, on Heroku. You can do this with the toolbelt:

```
> heroku config:set twitterAppId=appId
```

Alternatively, you can go to your dashboard and configure the environment in the settings tab:



The screenshot shows the 'Config Vars' section of the Heroku dashboard. On the left, a sidebar titled 'Config Variables' explains that config vars change the way your app behaves. It notes that some addons come with their own config vars. On the right, the 'Config Vars' table lists four variables:

Config Var	Value
JAVA_OPTS	-Xmx384m -Xss512k -XX:+UseCompre
REDIS_URL	[REDACTED]
twitterAppId	[REDACTED]
twitterAppSecret	[REDACTED]

An 'Edit' button is located in the top right corner of the table.



Visit <https://devcenter.heroku.com/articles/config-vars> for more information.



Running your application

It is now time to run our application on Heroku!

If you haven't already done so, commit all your changes to your master branch. Now, simply push your master branch to the heroku remote with `git push heroku master`. This will download all the dependencies and build your application from scratch, so it can take a little time:

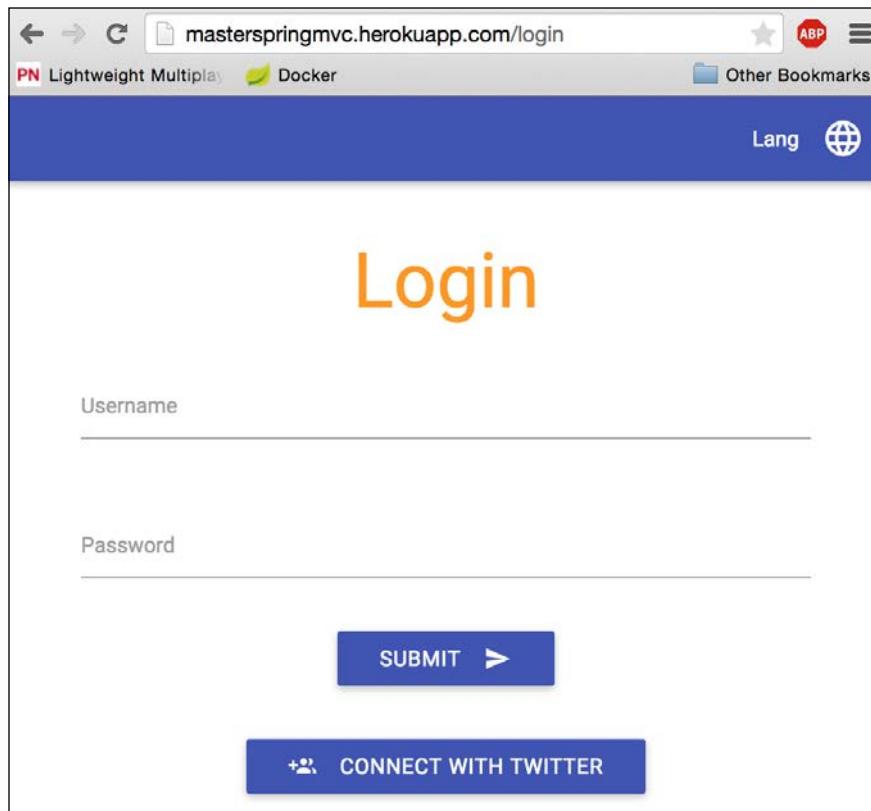
```
> git push heroku master
Counting objects: 1176, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (513/513), done.
Writing objects: 100% (1176/1176), 645.63 KiB | 0 bytes/s, done.
Total 1176 (delta 485), reused 1176 (delta 485)
remote: Compressing source files... done.
remote: Building source:
remote:
```

```
remote: -----> Gradle app detected
remote: -----> Installing OpenJDK 1.8... done
remote: -----> Building Gradle app...
remote:           WARNING: The Gradle buildpack is currently in Beta.
remote: -----> executing ./gradlew stage
remote:           Downloading https://services.gradle.org/distributions/
gradle-2.3-all.zip

...
remote:           :check
remote:           :build
remote:           :stage
remote:
remote:           BUILD SUCCESSFUL
remote:
remote:           Total time: 2 mins 36.215 secs
remote: -----> Discovering process types
remote:           Procfile declares types -> web
remote:
remote: -----> Compressing... done, 130.1MB
remote: -----> Launching... done, v4
remote:           https://appname.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy.... done.
To https://git.heroku.com/appname.git
 * [new branch]      master -> master
```

Once the application has been built, it will automatically run. Type `heroku logs` to see the latest logs or `heroku logs -t` to tail them.

You can see your application running in the console and if all goes as planned, you will be able to connect to `http://yourapp.herokuapp.com`. As shown in the following screenshot:



We are live! It's time to tell your friends!

Activating Redis

To activate Redis in our application, we can choose between a few alternatives. The Heroku Redis add-on is the beta version. It is entirely free with 20 MB of storage, analytics, and logs.



Visit <https://elements.heroku.com addons/heroku-redis> for more details.



At this stage, you will have to provide your credit card details to proceed.

To install the Redis add-on for your application, type the following:

```
heroku addons:create heroku-redis:test
```

Now, that we have activated the add-on, an environment variable called `REDIS_URL` will be available when our application will be running on Heroku.

You can check that the variable is defined with the `heroku config` command:

```
> heroku config
=== masterspringmvc Config Vars
JAVA_OPTS:          -Xmx384m -Xss512k -XX:+UseCompressedOops
REDIS_URL:          redis://x:xxx@ec2-xxx-xx-xxx-xxx.eu-west-1.compute.amazonaws.com:6439
```

Since the `RedisConnectionFactory` class does not understand URIs, we need to tweak it a little bit:

```
@Configuration
@Profile("redis")
@EnableRedisHttpSession
public class RedisConfig {

    @Bean
    @Profile("heroku")
    public RedisConnectionFactory redisConnectionFactory() throws
    URISyntaxException {
        JedisConnectionFactory redis = new JedisConnectionFactory();

        String redisUrl = System.getenv("REDIS_URL");
        URI redisUri = new URI(redisUrl);
        redis.setHostName(redisUri.getHost());
        redis.setPort(redisUri.getPort());
        redis.setPassword(redisUri.getUserInfo().split(":", 2)[1]);

        return redis;
    }

    @Bean
    @Profile({"cloud", "heroku"})
    public static ConfigureRedisAction configureRedisAction() {
        return ConfigureRedisAction.NO_OP;
    }
}
```

We now have two Heroku-specific beans in the `RedisConfig` class. These beans will only be active if both the `redis` and `heroku` profiles are active.

Note that we also deactivated some Spring Session configuration.

Spring Session will normally listen to events associated to destroyed session keys via the Redis Pub/Sub interface.

It will automatically try to configure the Redis environment to activate listeners on startup. In a secured environment like ours, adding listeners is not permitted unless you have an admin access.

These `redis` listeners are not really important in our case, so we can safely disable this behavior. For more information, visit <http://docs.spring.io/spring-session/docs/current/reference/html5/#api-redisoperationsessionrepository-sessiondestroyedevent>.

We need to modify our `Procfile` file so that Heroku runs our application with the `redis` profile:

```
web: java -Dserver.port=$PORT -Dspring.profiles.  
active=heroku,redis,prod -jar app.jar
```

Commit your change and push the code to Heroku.

Improving your application

We have a pretty good application deployed online but it's not uber useful nor original until you make it so.

Try to make it better and more personal. Once you're proud of your achievement, tweet your application URL with the `#masterspringmvc` hashtag on Twitter.

Try to push the best application possible. There is so much that we didn't do. Here are some ideas:

- Delete users' old pictures to avoid keeping unused pictures
- Use Twitter authentication information to fill the user profile
- Interact with the user's account
- See real-time searches happening on your app with a web socket channel

Let your imagination fly!

My version of the application is deployed on <http://masterspringmvc.herokuappapp.com>. I will improve some details to make the application a little more reactive. Try to spot the differences!

Summary

Deploying our application on a cloud provider is really straightforward as it is a runnable jar, thanks to Spring Boot. Cloud deployment is very affordable nowadays and deploying a Java application has become almost too easy.

With sessions backed by Redis, we laid the basics of a scalable application. Indeed, we can effortlessly add multiple servers behind a load balancer and absorb high traffic on demand.

The only thing that is not scalable is our WebSocket that will need additional work to run on top of a message broker, such as Rabbit MQ.

I can certainly remember a time where finding a host running a Tomcat was rare and pricey. Those days are long gone and the future belongs to web developers, so make it happen!

In the next chapter, we will see what we can do to make our application even better, discuss the technologies we haven't covered, talk about the Spring ecosystem in general, and the challenges of modern web applications.

10

Beyond Spring Web

In this chapter, we'll see how far we have come, the problems we've solved, and the ones left to be addressed.

We will talk about the Spring ecosystem in general, and persistence, deployment, and Single Page Applications in particular.

The Spring ecosystem

From the Web to data, Spring is a comprehensive ecosystem aiming to resolve all sorts of problems in a modular way:



Check out the Spring IO platform at <https://spring.io/platform>.

Core

At the core of the Spring framework, there is obviously a dependency injection mechanism.

We only scratched the surface of the security features and the great integration of the framework with Groovy.

Execution

We saw in detail what Spring Boot is about -- bringing simplicity and cohesion to a vast network of subprojects.

It allows you to focus on what really matters, that is, your business code.

The Spring XD project is also really interesting. Its goal is to provide tools to process, analyze, and transform or export your data, and has a clear focus on big data. For more information, visit <http://projects.spring.io/spring-xd/>.

Data

One of the things we haven't looked at while developing our application is how to store data in a database. In Pivotal's reference architecture, there is a tier devoted to both relational data and non-relational (NoSQL) data.

The Spring ecosystem has provided a lot of interesting solutions under the label `spring-data`, which can be found at <http://projects.spring.io/spring-data/>.

We glanced at Spring Data Redis when we built the cache but there is much more to Spring Data.

The basic concepts are shared among all the Spring Data projects, such as the template API, which is an abstraction to retrieve and store objects from a persistence system.

Spring Data JPA (<http://projects.spring.io/spring-data-jpa/>) and Spring Data Mongo (<http://projects.spring.io/spring-data-mongodb/>) are some of the most well known Spring Data projects. They let you operate on entities through repositories, simple interfaces that provide facilities to create queries, persisting objects, and so on.

Petri Kainulainen (<http://www.petrikainulainen.net/spring-data-jpa-tutorial/>) has a lot of thorough examples on Spring Data. It does not use the facilities that Spring Boot provides but you should be able to get started quite easily with guides, such as the one available at <https://spring.io/guides/gs/accessing-data-jpa/>.

Spring Data REST is also a magical project that will semiautomatically expose your entities through a RESTful API. Visit <https://spring.io/guides/gs/accessing-data-rest/> for a detailed tutorial.

Other noteworthy projects

Spring Integration (<http://projects.spring.io/spring-integration>) and Spring Reactor (<http://projectreactor.io>) are also two of my favorite Spring projects.

Spring Reactor is the implementation of reactive streams by Pivotal. The idea is to provide fully nonblocking IO on the server side.

Spring Integration, on the other hand, focuses on Enterprise Integration Patterns and lets you design channels to load and transform data coming from heterogeneous systems.

A good, and simple, example of what you can accomplish with channels can be seen here: http://lmivan.github.io/contest/#_spring_boot_application.

If you have heterogeneous and/or complex subsystems with which your application has to communicate, it is definitely worth taking a look at.

The last project in the Spring ecosystem we haven't is Spring Batch, a really useful abstraction for processing high volumes of data for the daily operations of enterprise systems.

The deployment

Spring Boot provides the ability to run and distribute your Spring application as a simple JAR and is a wonderful success in that regard.

It is, without a doubt, a step in the right direction, but sometimes your web application isn't the only thing you want to deploy.

When dealing with a complex system with multiple servers and datasources, the work of the operation team can become quite a headache.

Docker

Who hasn't heard about Docker? It is the new cool kid in the container world and has become quite a success, thanks to its vibrant community.

The ideas behind Docker are not new, it leverages LinuX Containers (LXC) and cgroups to provide a fully isolated environment for applications to run in.

You can find a tutorial on the Spring website that will guide you through your first steps with Docker at <https://spring.io/guides/gs/spring-boot-docker>.

Pivotal Cloud Foundry has been using container technology for years in their container manager called Warden. They recently moved to Garden, an abstraction that supports not only Linux containers, but also Windows containers.

Garden is part of the latest release of Cloud Foundry (called Diego) that also allows Docker images as units of deployment.

A developer version of Cloud Foundry has also been released under the name Lattice, which can be found at <https://spring.io/blog/2015/04/06/lattice-and-spring-cloud-resilient-sub-structure-for-your-cloud-native-spring-applications>.

If you want to test containers without the hassles of the command line, I recommend that you look at Kitematic. With this, you can run a Jenkins container or a MongoDB without installing the binaries on your system. Visit <https://kitematic.com/> for more information on Kitematic.

Another tool in the Docker ecosystem that's worth mentioning is Docker Compose. It allows you to run and link multiple containers with a single configuration file.

Refer to <http://java.dzone.com/articles/spring-session-demonstration> for a good example of a Spring Boot application composed of two web servers, a Redis to store users' sessions, and an Nginx instance to do the load balancing. Of course, there is much more to learn about Docker Swarm, which will allow you to scale your application with a simple command, and Docker Machine, which will create Docker hosts for you on any machine, including Cloud providers.

Google Kuberntes and Apache Mesos are also great examples of distributed systems that benefit greatly from Docker containers.

Single Page Applications

Most of today's web applications are written in JavaScript. Java is relegated to the backend and has the important role of dealing with data and business rules. However, much of the GUI stuff is now happening on the client side.

There is a good reason for that in terms of responsiveness and user experience, but those applications add extra complexity.

Developers now have to be fluent in both Java and JavaScript and the number of frameworks can be a little overwhelming at first.

The players

If you want to dig deeper into JavaScript, I would highly recommend Dave Syer's tutorial with Spring and AngularJS, which is available at <https://spring.io/guides/tutorials/spring-security-and-angular-js>.

Choosing a JavaScript MVC framework can be a little difficult too. AngularJS has had the favor of the Java community for years but people seem to be moving away from it. For more information, visit <https://gist.github.com/tdd/5ba48ba5a2a179f2d0fa>.

Other alternatives include the following:

- **BackboneJS:** This is a really simple MVC framework that sits on top of Underscore and jQuery.
- **Ember:** This is a comprehensive system that provides more facilities for interacting with data and more.
- **React:** This is the newest project from Facebook. It has a new and very interesting philosophy for dealing with views. Its learning curve is quite steep, but it is a very interesting system to look at in terms of designing a GUI framework.

React is my favorite project right now. It lets you focus on the view and its one-way data flow makes it easy to reason with the state of your application. However, it is still in version 0.13. This makes it both very interesting, as the vibrant community always comes up with new solutions and ideas, and somewhat disturbing, as the road ahead stills seems long even after more than 2 years of open source development. Visit <https://facebook.github.io/react/blog/2014/03/28/the-road-to-1.0.html> for information on "The Road to 1.0".

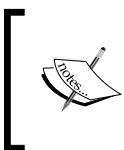
The future

I see a lot of Java developers ranting about the permissiveness of JavaScript and having a hard time dealing with the fact that it is not a strongly typed language.

There are other alternatives, such as **Typescript** (<http://www.typescriptlang.org/>), which are really interesting and provide the things that we, Java developers, have always used to make our lives simpler: interfaces, classes, helpful support in IDE, and autocompletion.

A lot of people place bets on the next version (2.0) of Angular that will quite notoriously break everything. I think it's for the best. Their collaboration with Microsoft's team that makes Typescript is really unique.

Most JEE developers will smile when they hear that one of the big new features of ECMAScript, that allows the development of this new framework, is decorators, some kind of annotation mechanism:



To learn about the difference between annotation and decorators, visit <http://blog.thoughtram.io/angular/2015/05/03/the-difference-between-annotations-and-decorators.html>.



JavaScript is evolving quickly and ECMAScript 6 has a lot of interesting features that makes it a really advanced and sophisticated language. Don't miss the boat; take a look at <https://github.com/lukehoban/es6features> before it's too late!

The web component specification is also a game changer. The goal is to provide reusable UI components, and both the React team and the Angular 2 teams have plans to interface with it. Google has developed an interesting project on top of web components called Polymer that is now in version 1.0.



Refer to the article at <http://ng-learn.org/2014/12/Polymer/> to learn more about the state of these projects.



Going stateless

When dealing with a JavaScript client, relying on session cookies isn't the best option. Most applications choose to go completely stateless and identify clients with a token.

If you want to stick with Spring Session, take a look at the `HeaderHttpSessionStrategy` class. It has an implementation that sends and retrieves sessions with an HTTP header. An example of this can be found at <https://drissamri.be/blog/2015/05/21/spring-security-and-spring-session/>.

Summary

The Spring ecosystem is wide and has a lot to offer to modern web application developers.

It is hard to find a problem that hasn't been addressed by one of the Spring projects.

Time to say good bye! I hope you enjoyed our little journey with Spring MVC and that it will help you develop with pleasure and create amazing projects, at work or in your spare time.

Index

A

- acceptance tests**
 - about 190-194, 213
 - Gradle, configuration 213, 214
 - with FluentLenium 215-221
 - with Groovy 225
- Anemic Domain Model**
 - about 35, 36
 - URL 35
- API versioning**
 - about 133
 - reference link 134
- application cache**
 - creating 240-245
 - distributed cache 247, 248
 - invalidation 246
 - reference link 245
- async methods**
 - reference link 253
 - using 248-253
- authentication**
 - testing 211, 212
- authorized URLs**
 - authenticating 165, 166
- authorized users**
 - authenticating 163-165

B

- BackboneJS** 287
- basic authentication**
 - about 161
 - configuring 162

- for authorized URLs 165, 166
- for authorized users 163, 164
- Thymeleaf security tags, using 167, 168
- URL 161

C

- cache control**
 - about 238
 - configuring 239
- check point** 158
- client validation, profile page**
 - enabling 88, 89
 - reference link 88
- Cloud Foundry**
 - about 264
 - CLI tools, installing 265, 266
 - URL 264
- code testing**
 - acceptance tests 190, 191
 - benefits 190
 - unit tests 190, 191
- constructor injection**
 - about 111
 - URL 111
- continuous integration**
 - reference link 189
- Create Read Update Delete (CRUD)** 131
- Cross Site Request Forgery (CSRF)**
 - about 166
 - URL 166
- custom error page**
 - creating 112, 113

D

Data Transfer Object (DTO) 66
deployment
about 285
Docker 286
Dispatcher Servlet
architecture 42, 43
distributed cache
configuring 247, 248
distributed sessions
about 180
setting up 180-182
Docker
about 286
URL 286
documentation
with Swagger 155, 156
Domain Driven Design (DDD) 35

E

embedded Servlet container (Tomcat)
configuration 28, 29
HTTP port, setting 30
other configurations 31, 32
SSL configuration 30
Ember 287
encoding configuration 26-28
error handling 26-28
error messages
translating 108
ETags
about 254
generating 254
using 255-258
exception handling 148

F

file upload
about 93
check point 127
errors, handling 104-107
implementation 119-126
profile picture, uploading 93-98

uploaded images, displaying on
web page 98, 99
uploaded picture, displaying 102-104
upload properties, managing 99-102
FluentLenium
about 215
Page Objects 221-224
URL 217
used, for acceptance tests 215-221

G

Geb
about 229
Page Objects 230-234
reference link 234
used, for integration tests 229, 230
Git
about 95
empty directory 95
installing 11, 12
URL 11
web application, saving 11, 12

Gradle
about 12
configuration 213, 214
installing 12-14
JAR file 15
running 275
URL 275
Groovy
about 225
acceptance tests 225
URL 225
Groovy Development Kit (GDK) 225
Gzipping
about 238
reference link 238

H

HandlerMapping 42
Heroku
about 265
command-line tools, installing 273, 274
Gradle, running 275

Heroku profile, creating 276, 277
Procfile, running 276
Redis, activating 279-281
web application, deploying 273
web application, executing 277, 278
web application, setting up 274

Heroku Redis add-on

URL 279

host

Cloud Foundry 264
Heroku 265
OpenShift 264, 265
selecting 263

HTTP codes

about 134, 135
URL 135

httpie 138

HTTP port

setting 30

HTTP sessions

about 109
profile, storing 109-112

HTTP verbs

DELETE 131
GET 131
HEAD 131
OPTIONS 131
PATCH 131
POST 131
PUT 131

HyperText As The Engine Of Application State (HATEOAS) 132

I

iconmonstr

URL 99

integration tests

with Geb 229, 230

IntelliJ

about 8

project, creating 8

Interceptors 81

internationalization (i18n)

about 78, 79

application text, translating 82-84

data list, handling in form 84-87

locale, modifying 80-82

J

Java 8

date time API, URL 66
lambdas 51
streams 51

JSON output

customizing 139-142

JSR-310 Module

about 142
URL 143

L

lambdas, Java 8 51

layouts

using 55, 56

locale configuration 23-25

login form

designing 169-173

M

material design

with WebJars 52-54

Materialize

URL 33

matrix variables

URL mapping 114-119

mocks

about 199

and stubs, selecting between 204

creating, Mockito used 199-201

reference link 204

Multipurpose Internet Mail Extensions

(MIME) 98

MVC architecture

about 34, 35

Anemic Domain Model 35, 36

best practice 35

Controller 34

critics 35

Model 34

sagan project 36

View 34

N

navigation

Forward option 59
Redirect option 59
using 57-61

O

OpenShift

about 264
URL 265

P

Page Objects

with FluentLenium 221-224
with Geb 230-234

PhantomJS

URL 217

Pivotal Web Services (PWS)

web application, deploying 265

Plain Old Java Object (POJO) 35, 66

Procfile

running 276

production profile

configuring 237

profile page

about 63

check point 90

client validation, enabling 88, 89

creating 64-71

validation, adding 71-73

Q

query parameters 45

R

React 287

Redis

activating 272, 273, 279-281

URL 181

Representational State Transfer (REST)

about 129

controllers, unit testing 204-210

RESTful API, debugging

about 138
httpie 138
JSON formatting extension 138
RESTful client, in browser 138

RESTful web service, properties

cacheable 129
client-server 129
layered 129
stateless 129
uniform interface 129

Richardson's maturity model

about 130
level 0 - HTTP 130
level 1 - Resources 130
level 2 - HTTP verbs 131, 132
level 3 - Hypermedia controls 132, 133

S

SearchApiController class

creating, in search.api package 136, 137

Secure Sockets Layer. See SSL

security headers

about 165
URL 165

self-signed certificate

generating 183

Single Page Applications

about 287
future enhancements 288
recommendations 287
reference link 289
stateless option 289

Sockjs 258

Spock

used, for unit tests 225-228

Spring

core 284
data 284
ecosystem 283
execution 284
noteworthy projects 285
URL 283

Spring Boot

about 18
configuration 19-22

initializing 18
locale configuration 23-25
logging in 68
static resources 23-25
URL 68
view resolver 23-25

Spring Data JPA
 URL 284

Spring Data Mongo
 URL 284

Spring Data REST
 URL 285

Spring Expression Language (SpEL)
 about 44, 45
 data, obtaining with request
 parameter 45, 46
 URL 45

Springfox
 URL 157

Spring Integration
 URL 285

Spring MVC
 architecture 42
 web application 42

Spring MVC 1-0-1
 about 37
 reference link 37

Spring Reactor
 about 285
 URL 285

Spring Security 4
 reference link 213

Spring Social
 about 48
 URL 48

Spring Social Twitter project
 application, registering 47, 48
 creating 47
 setting up 48
 Twitter, accessing 49, 50
 URL 33

Spring Tool Suite (STS)
 about 2
 Gradle support, downloading 2
 Groovy Eclipse plugin, installing 2, 3
 project, starting 4-7
 URL 2

SSL
 about 183
 configuration 30
 creating 184
 creating, behind secured server 186
 creating, for http and https
 channels 184, 185
 reference link 180, 186
 self-signed certificate, generating 183

start.Spring.io
 about 9
 command line 9, 10
 project, creating 9
 URL 9

static resources
 about 23-25
 configuration 24, 25

status code
 200 OK 148
 400 Bad Request 148
 404 Not Found 148
 405 Method not Supported 148
 500 Server Error 148
 about 148
 with exception 151-155
 with ResponseEntity 149, 150

streams, Java 8 51

stubs
 about 199
 and mocks, selecting between 204
 creating, for testing beans 201, 202

Swagger 155

T

test-driven development (TTD) 192

th:each tag 50

Thymeleaf
 about 38
 page, adding 40
 reference link 38, 168
 using 38
 security tags, using 167, 168

tools
 about 194
 AssertJ 194
 DbUnit 194

JUnit 194
Mockito 194
Spock 194

Twitter authentication

coding 178-180
setting up 174
social authentication, setting up 175-178
URL 174

TypeScript

about 288
URL 288

U

unit tests

about 190-193
REST controllers 204-210
tools 194
with Spock 225-228
writing 195-198

URL mapping

with matrix variables 114-119

user management API 14-148

V

validation, profile page

adding 71-73
custom annotation, defining 77
reference link 72
validation messages, customizing 73-76

validators

reference link 89

view resolver

about 23-25
configuration 24

W

war file

URL 6

web application

assembling 267-271
Cloud Foundry CLI tools,
installing 265, 266
code 16-18
data, displaying 43, 44
deploying, on Heroku 273
deploying, to Pivotal Web Services
(PWS) 265

Dispatcher Servlet 42, 43

executing, on Heroku 277, 278

Gradle, using 12-16

improving 281

Redis, activating 272, 273

saving, in Git 11, 12

setting up, on Heroku 274

WebJars

layouts, using 55, 56

navigation, using 57-61

TweetController, using 61, 62

used, for material design 52-54

WebSocket

about 258

reference link 261

using 258-261

X

XML

generating 157, 158



Thank you for buying Mastering Spring MVC 4

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at www.packtpub.com.

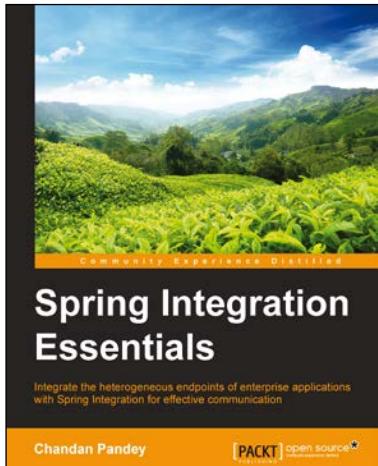
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

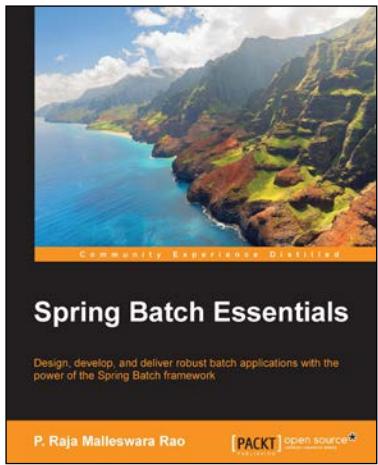


Spring Integration Essentials

ISBN: 978-1-78398-916-4 Paperback: 198 pages

Integrate the heterogeneous endpoints of enterprise applications with Spring Integration for effective communication

1. Tackle the challenges of enterprise integration and experience how Spring integration can transform these challenges into solutions.
2. Develop the skills necessary to apply integration patterns for heterogeneous enterprise endpoint communication and select the best and most suited Spring components.
3. Reuse working code snippets that can be handy for integration scenarios such as Twitter, e-mail, FTP, databases, and many others.



Spring Batch Essentials

ISBN: 978-1-78355-337-2 Paperback: 148 pages

Design, develop, and deliver robust batch applications with the power of the Spring Batch framework

1. Leverage the POJO-based development approach to create comprehensive batch applications.
2. Customize the batch job components with the flexible XML, Annotations, and Expression Language-based configuration.
3. Enable high-volume and high-performance batch jobs through optimization and partitioning techniques.

Please check www.PacktPub.com for information on our titles

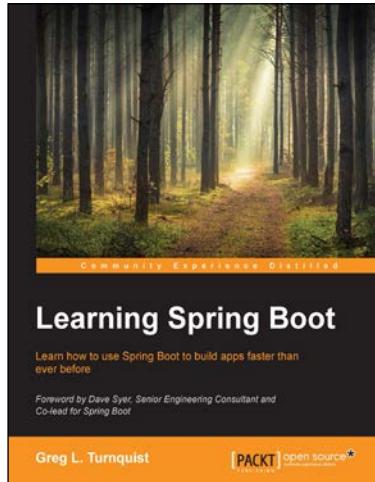


Learning Heroku Postgres

ISBN: 978-1-78217-345-8 Paperback: 164 pages

Efficiently design, implement, and manage a successful PostgreSQL database with Heroku

1. Manage and optimize your PostgreSQL database with Heroku Postgres.
2. Secure your database with rollback, followers and forks functionalities.
3. A step-by-step tutorial with examples to help you get to grips with proficiency in Heroku Postgres database.



Learning Spring Boot

ISBN: 978-1-78439-302-1 Paperback: 252 pages

Learn how to use Spring Boot to build apps faster than ever before

1. Create Spring-powered, production-grade applications and services with minimal fuss.
2. Support multiple environments with one artifact, and add production-grade support with features like custom metrics to track the number of messages published and consumed.
3. Each chapter introduces a different area that Spring Boot tackles and explains how to tweak your apps through different properties.

Please check www.PacktPub.com for information on our titles