



Spring Boot IN ACTION

Craig Walls

FOREWORD BY Andrew Glover

 MANNING

Spring Boot in Action

CRAIG WALLS



MANNING
Shelter Island

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact


Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

©2016 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

© Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

 Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor: Cynthia Kane
Technical development editor: Robert Casazza
Copyeditor: Andy Carroll
Proofreader: Corbin Collins
Technical proofreader: John Guthrie
Typesetter: Gordan Salinovic
Cover designer: Marija Tudor

ISBN 9781617292545

Printed in the United States of America

1 2 3 4 5 6 7 8 9 10 – EBM – 20 19 18 17 16 15

contents

foreword vii
preface ix
about this book xii
acknowledgments xv

1 **Bootstarting Spring** 1

1.1 Spring rebooted 2

Taking a fresh look at Spring 2 ▪ *Examining Spring Boot essentials* 4 ▪ *What Spring Boot isn't* 7

1.2 Getting started with Spring Boot 8

Installing the Spring Boot CLI 8 ▪ *Initializing a Spring Boot project with Spring Initializr* 12

1.3 Summary 22

2 **Developing your first Spring Boot application** 23

2.1 Putting Spring Boot to work 24

Examining a newly initialized Spring Boot project 26 ▪ *Dissecting a Spring Boot project build* 30

2.2 Using starter dependencies 33

Specifying facet-based dependencies 34 ▪ *Overriding starter transitive dependencies* 35

- 2.3 Using automatic configuration 37
 - Focusing on application functionality* 37 ▪ *Running the application* 43 ▪ *What just happened?* 45
- 2.4 Summary 48

3 Customizing configuration 49

- 3.1 Overriding Spring Boot auto-configuration 50
 - Securing the application* 50 ▪ *Creating a custom security configuration* 51 ▪ *Taking another peek under the covers of auto-configuration* 55
- 3.2 Externalizing configuration with properties 57
 - Fine-tuning auto-configuration* 58 ▪ *Externally configuring application beans* 64 ▪ *Configuring with profiles* 69
- 3.3 Customizing application error pages 71
- 3.4 Summary 74

4 Testing with Spring Boot 76

- 4.1 Integration testing auto-configuration 77
- 4.2 Testing web applications 79
 - Mocking Spring MVC* 80 ▪ *Testing web security* 83
- 4.3 Testing a running application 86
 - Starting the server on a random port* 87 ▪ *Testing HTML pages with Selenium* 88
- 4.4 Summary 90

5 Getting Groovy with the Spring Boot CLI 92

- 5.1 Developing a Spring Boot CLI application 93
 - Setting up the CLI project* 93 ▪ *Eliminating code noise with Groovy* 94 ▪ *What just happened?* 98
- 5.2 Grabbing dependencies 100
 - Overriding default dependency versions* 101 ▪ *Adding dependency repositories* 102
- 5.3 Running tests with the CLI 102
- 5.4 Creating a deployable artifact 105
- 5.5 Summary 106

6 *Applying Grails in Spring Boot* 107

- 6.1 Using GORM for data persistence 108
- 6.2 Defining views with Groovy Server Pages 113
- 6.3 Mixing Spring Boot with Grails 3 115
 - Creating a new Grails project* 116 ▪ *Defining the domain* 118
 - Writing a Grails controller* 119 ▪ *Creating the view* 120
- 6.4 Summary 123

7 *Taking a peek inside with the Actuator* 124

- 7.1 Exploring the Actuator's endpoints 125
 - Viewing configuration details* 126 ▪ *Tapping runtime metrics* 133
 - Shutting down the application* 139 ▪ *Fetching application information* 140
- 7.2 Connecting to the Actuator remote shell 141
 - Viewing the autoconfig report* 142 ▪ *Listing application beans* 143
 - Watching application metrics* 144 ▪ *Invoking Actuator endpoints* 145
- 7.3 Monitoring your application with JMX 146
- 7.4 Customizing the Actuator 148
 - Changing endpoint IDs* 148 ▪ *Enabling and disabling endpoints* 149
 - Adding custom metrics and gauges* 149 ▪ *Creating a custom trace repository* 153 ▪ *Plugging in custom health indicators* 155
- 7.5 Securing Actuator endpoints 156
- 7.6 Summary 159

8 *Deploying Spring Boot applications* 160

- 8.1 Weighing deployment options 161
- 8.2 Deploying to an application server 162
 - Building a WAR file* 162 ▪ *Creating a production profile* 164
 - Enabling database migration* 168
- 8.3 Pushing to the cloud 173
 - Deploying to Cloud Foundry* 173 ▪ *Deploying to Heroku* 177
- 8.4 Summary 180

appendix A Spring Boot Developer Tools 181

appendix B Spring Boot starters 188

appendix C Configuration properties 195

appendix D Spring Boot dependencies 232

index 243

foreword

In the spring of 2014, the Delivery Engineering team at Netflix set out to achieve a lofty goal: enable end-to-end global continuous delivery via a software platform that facilitates both extensibility and resiliency. My team had previously built two different applications attempting to address Netflix’s delivery and deployment needs, but both were beginning to show the telltale signs of monolith-ness and neither met the goals of flexibility and resiliency. What’s more, the most stymieing effect of these monolithic applications was ultimately that we were unable to keep pace with our partner’s innovation. Users had begun to move around our tools rather than with them.

It became apparent that if we wanted to provide real value to the company and rapidly innovate, we needed to break up the monoliths into small, independent services that could be released at will. Embracing a microservice architecture gave us hope that we could also address the twin goals of flexibility and resiliency. But we needed to do it on a credible foundation where we could count on real concurrency, legitimate monitoring, reliable and easy service discovery, and great runtime performance.

With the JVM as our bedrock, we looked for a framework that would give us rapid velocity and steadfast operationalization out of the box. We zeroed in on Spring Boot.

Spring Boot makes it effortless to create Spring-powered, production-ready services without a lot of code! Indeed, the fact that a simple Spring Boot Hello World application can fit into a tweet is a radical departure from what the same functionality required on the JVM only a few short years ago. Out-of-the-box nonfunctional features like security, metrics, health-checks, embedded servers, and externalized configuration made Boot an easy choice for us.

Yet, when we embarked on our Spring Boot journey, solid documentation was hard to come by. Relying on source code isn't the most joyful manner of figuring out how to properly leverage a framework's features.

It's not surprising to see the author of Manning's venerable *Spring in Action* take on the challenge of concisely distilling the core aspects of working with Spring Boot into another cogent book. Nor is it surprising that Craig and the Manning crew have done another tremendously wonderful job! *Spring Boot in Action* is an easily readable book, as we've now come to expect from Craig and Manning.

From chapter 1's attention-getting introduction to Boot and the now legendary 90ish-character tweetable Boot application to an in-depth analysis of Boot's Actuator in chapter 7, which enables a host of auto-magical operational features required for any production application, *Spring Boot in Action* leaves no stone unturned. Indeed, for me, chapter 7's deep dive into the Actuator answered some of the lingering questions I've had in the back of my head since picking up Boot well over a year ago. Chapter 8's thorough examination of deployment options opened my eyes to the simplicity of Cloud Foundry for cloud deployments. One of my favorite chapters is chapter 4, where Craig explores the many powerful options for easily testing a Boot application. From the get-go, I was pleasantly surprised with some of Spring's testing features, and Boot takes advantage of them nicely.

As I've publicly stated before, Spring Boot is just the kind of framework the Java community has been seeking for over a decade. Its easy-to-use development features and out-of-the-box operationalization make Java development fun again. I'm pleased to report that Spring and Spring Boot are the foundation of Netflix's new continuous delivery platform. What's more, other teams at Netflix are following the same path because they too see the myriad benefits of Boot.

It's with equal parts excitement and passion that I absolutely endorse Craig's book as the easy-to-digest and fun-to-read Spring Boot documentation the Java community has been waiting for since Boot took the community by storm. Craig's accessible writing style and sweeping analysis of Boot's core features and functionality will surely leave readers with a solid grasp of Boot (along with a joyful sense of awe for it).

Keep up the great work Craig, Manning Publications, and all the brilliant developers who have made Spring Boot what it is today! Each one of you has ensured a bright future for the JVM.

ANDREW GLOVER
MANAGER, DELIVERY ENGINEERING AT NETFLIX

preface

At the 1964 New York World's Fair, Walt Disney introduced three groundbreaking attractions: "it's a small world," "Great Moments with Mr. Lincoln," and the "Carousel of Progress." All three of these attractions have since moved into Disneyland and Walt Disney World, and you can still see them today.

My favorite of these is the Carousel of Progress. Supposedly, it was one of Walt Disney's favorites too. It's part ride and part stage show where the seating area rotates around a center area featuring four stages. Each stage tells the story of a family at different time periods of the 20th century—the early 1900s, the 1920s, the 1940s, and recent times—highlighting the technology advances in that time period. The story of innovation is told from a hand-cranked washing machine, to electric lighting and radio, to automatic dishwashers and television, to computers and voice-activated appliances.

In every act, the father (who is also the narrator of the show) talks about the latest inventions and says "It can't get any better," only to discover that, in fact, it does get better in the next act as technology progresses.

Although Spring doesn't have quite as long a history as that displayed in the Carousel of Progress, I feel the same way about Spring as "Progress Dad" felt about the 20th century. Each and every Spring application seems to make the lives of developers so much better. Just looking at how Spring components are declared and wired together, we can see the following progression over the history of Spring:

- When Spring 1.0 hit the scene, it completely changed how we develop enterprise Java applications. Spring dependency injection and declarative transactions meant no more tight coupling of components and no more heavyweight EJBs. It couldn't get any better.
- With Spring 2.0 we could use custom XML namespaces for configuration, making Spring itself even easier to use with smaller and easier to understand configuration files. It couldn't get any better.
- Spring 2.5 gave us a much more elegant annotation-oriented dependency-injection model with the `@Component` and `@Autowired` annotations, as well as an annotation-oriented Spring MVC programming model. No more explicit declaration of application components, and no more subclassing one of several base controller classes. It couldn't get any better.
- Then with Spring 3.0 we were given a new Java-based configuration alternative to XML that was improved further in Spring 3.1 with a variety of `@Enable`-prefixed annotations. For the first time, it became realistic to write a complete Spring application with no XML configuration whatsoever. It couldn't get any better.
- Spring 4.0 unleashed support for conditional configuration, where runtime decisions would determine which configuration would be used and which would be ignored based on the application's classpath, environment, and other factors. We no longer needed to write scripts to make those decisions at build time and pick which configuration should be included in the deployment. How could it possibly get any better?

And then came Spring Boot. Even though with each release of Spring we thought it couldn't possibly get any better, Spring Boot proved that there's still a lot of magic left in Spring. In fact, I believe Spring Boot is the most significant and exciting thing to happen in Java development in a long time.

Building upon previous advances in the Spring Framework, Spring Boot enables automatic configuration, making it possible for Spring to intelligently detect what kind of application you're building and automatically configure the components necessary to support the application's needs. There's no need to write explicit configuration for common configuration scenarios; Spring will take care of it for you.

Spring Boot starter dependencies make it even easier to select which build-time and runtime libraries to include in your application builds by aggregating commonly needed dependencies. Spring Boot starters not only keep the dependencies section of your build specifications shorter, they keep you from having to think too hard about the specific libraries and versions you need.

Spring Boot's command-line interface offers a compelling option for developing Spring applications in Groovy with minimal noise or ceremony common in Java applications. With the Spring Boot CLI, there's no need for accessor methods, access modifiers such as `public` or `private`, semicolons, or the `return` keyword. In many cases, you can even eliminate `import` statements. And because you run the application as scripts from the command line, you don't need a build specification.

Spring Boot's Actuator gives you insight into the inner workings of a running application. You can see exactly what beans are in the Spring application context, how Spring MVC controllers are mapped to paths, the configuration properties available to your application, and much more.

With all of these wonderful features enabled by Spring Boot, it certainly can't get any better!

In this book, you'll see how Spring Boot has indeed made Spring even better than it was before. We'll look at auto-configuration, Spring Boot starters, the Spring Boot CLI, and the Actuator. And we'll tinker with the latest version of Grails, which is based on Spring Boot. By the time we're done, you'll probably be thinking that Spring couldn't get any better.

If we've learned anything from Walt Disney's Carousel of Progress, it's that when we think things can't get any better, they inevitably do get better. Already, the advances offered by Spring Boot are being leveraged to enable even greater advances. It's hard to imagine Spring getting any better than it is now, but it certainly will. With Spring, there's always a great big beautiful tomorrow.

about this book

Spring Boot aims to simplify Spring development. As such, Spring Boot's reach stretches to touch everything that Spring touches. It'd be impossible to write a book that covers every single way that Spring Boot can be used, as doing so would involve covering every single technology that Spring itself supports. Instead, *Spring Boot in Action* aims to distill Spring Boot into four main topics: auto-configuration, starter dependencies, the command-line interface, and the Actuator. Along the way, we'll touch on a few Spring features as necessary, but the focus will be primarily on Spring Boot.

Spring Boot in Action is for all Java developers. Although some background in Spring could be considered a prerequisite, Spring Boot has a way of making Spring more approachable even to those new to Spring. Nevertheless, because this book will be focused on Spring Boot and will not dive deeply into Spring itself, you may find it helpful to pair it with other Spring materials such as *Spring in Action, Fourth Edition* (Manning, 2014).

Roadmap

Spring Boot in Action is divided into seven chapters:

- In chapter 1 you'll be given an overview of Spring Boot, including the essentials of automatic configuration, starter dependencies, the command-line interface, and the Actuator.
- Chapter 2 takes a deeper dive into Spring Boot, focusing on automatic configuration and starter dependencies. In this chapter, you'll build a complete Spring application using very little explicit configuration.

- Chapter 3 picks up where chapter 2 leaves off, showing how you can influence automatic configuration by setting application properties or completely overriding automatic configuration when it doesn't meet your needs.
- In chapter 4 we'll look at how to write automated integration tests for Spring Boot applications.
- In chapter 5 you'll see how the Spring Boot CLI offers a compelling alternative to conventional Java development by enabling you to write complete applications as a set of Groovy scripts that are run from the command line.
- While we're on the subject of Groovy, chapter 6 takes a look at Grails 3, the latest version of the Grails framework, which is now based on Spring Boot.
- In chapter 7 you'll see how to leverage Spring Boot's Actuator to dig inside of a running application and see what makes it tick. You'll see how to use Actuator web endpoints as well as a remote shell and JMX MBeans to peek at the internals of an application.
- Chapter 8 wraps things up by discussing various options for deploying your Spring Boot application, including traditional application server deployment and cloud deployment.

Code conventions and downloads

There are many code examples throughout this book. These examples will always appear in a fixed-width code font like this. Any class name, method name, or XML fragment within the normal text of the book will appear in code font as well. Many of Spring's classes and packages have exceptionally long (but expressive) names. Because of this, line-continuation markers (➞) may be included when necessary. Not all code examples in this book will be complete. Often I only show a method or two from a class to focus on a particular topic.

Complete source code for the applications found in the book can be downloaded from the publisher's website at www.manning.com/books/spring-boot-in-action.

Author Online

The purchase of *Spring Boot in Action* includes free access to a private web forum run by Manning Publications, where you can make comments about the book, ask technical questions, and receive help from the author and from other users. To access the forum and subscribe to it, point your web browser to www.manning.com/books/spring-boot-in-action. This page provides information on how to get on the forum once you are registered, what kind of help is available, and the rules of conduct on the forum.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the author some challenging questions lest his interest stray!

The Author Online forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

About the cover illustration

The figure on the cover of *Spring Boot in Action* is captioned "Habit of a Tartar in Kasan," which is the capital city of the Republic of Tatarstan in Russia. The illustration is taken from Thomas Jefferys' *A Collection of the Dresses of Different Nations, Ancient and Modern* (four volumes), London, published between 1757 and 1772. The title page states that these are hand-colored copperplate engravings, heightened with gum arabic. Thomas Jefferys (1719–1771) was called "Geographer to King George III." He was an English cartographer who was the leading map supplier of his day. He engraved and printed maps for government and other official bodies and produced a wide range of commercial maps and atlases, especially of North America. His work as a mapmaker sparked an interest in local dress customs of the lands he surveyed and mapped, which are brilliantly displayed in this collection.

Fascination with faraway lands and travel for pleasure were relatively new phenomena in the late eighteenth century, and collections such as this one were popular, introducing both the tourist as well as the armchair traveler to the inhabitants of other countries. The diversity of the drawings in Jefferys' volumes speaks vividly of the uniqueness and individuality of the world's nations some 200 years ago. Dress codes have changed since then, and the diversity by region and country, so rich at the time, has faded away. It is now often hard to tell the inhabitant of one continent from another. Perhaps, trying to view it optimistically, we have traded a cultural and visual diversity for a more varied personal life. Or a more varied and interesting intellectual and technical life.

At a time when it is hard to tell one computer book from another, Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by Jefferys' pictures.

acknowledgments

This book will show how Spring Boot can automatically deal with the behind-the-scenes stuff that goes into an application, freeing you to focus on the tasks that make your application unique. In many ways, this is analogous to what went into making this book happen. There were so many other people taking care of making things happen that I was free to focus on writing the content of the book. For taking care of the behind-the-scenes work at Manning, I'd like to thank Cynthia Kane, Robert Casazza, Andy Carroll, Corbin Collins, Kevin Sullivan, Mary Piergies, Janet Vail, Ozren Harlovic, and Candace Gillhoolley.

Writing tests help you know if your software is meeting its goals. Similarly, those who reviewed *Spring Boot in Action* while it was still being written gave me the feedback I needed to make sure that the book stayed on target. For this, my gratitude goes out to Aykut Acikel, Bachir Chihani, Eric Kramer, Francesco Persico, Furkan Kamaci, Gregor Zurowski, Mario Arias, Michael A. Angelo, Mykel Alvis, Norbert Kuchenmeister, Phil Whiles, Raphael Villela, Sam Kreter, Travis Nelson, Wilfredo R. Ronsini Jr., and William Fly. Special thanks to John Guthrie for a final technical review shortly before the manuscript went into production. And extra special thanks to Andrew Glover for contributing the foreword to my book.

Of course, this book wouldn't be possible or even necessary without the incredible work done by the talented members of the Spring team. It's amazing what you do, and I'm so excited to be part of a team that's changing how software is developed.

Many thanks to all of those involved in the No Fluff/Just Stuff tour, whether it be my fellow presenters or those who show up to hear us talk. The conversations we've had have in some small way contributed to how this book was formed.

A book like this would not be possible without an alphabet to compose into words. So, just as in my previous book, I'd like to take this opportunity to thank the Phoenicians for the invention of the first alphabet.

Last, but certainly not least...my love, devotion, and thanks go to my beautiful wife Raymie and my awesome girls, Maisy and Madi. Once again, you've tolerated another writing project. Now that it's done, we should go to Disney World. Whatdya say?

1 *Bootstarting Spring*

This chapter covers

- How Spring Boot simplifies Spring application development
- The essential features of Spring Boot
- Setting up a Spring Boot workspace

The Spring Framework has been around for over a decade and has found a place as the de facto standard framework for developing Java applications. With such a long and storied history, some might think that Spring has settled, resting on its laurels, and is not doing anything new or exciting. Some might even say that Spring is legacy and that it's time to look elsewhere for innovation.

Some would be wrong.

There are many exciting new things taking place in the Spring ecosystem, including work in the areas of cloud computing, big data, schema-less data persistence, reactive programming, and client-side application development.

Perhaps the most exciting, most head-turning, most game-changing new thing to come to Spring in the past year or so is Spring Boot. Spring Boot offers a new paradigm for developing Spring applications with minimal friction. With Spring Boot, you'll be able to develop Spring applications with more agility and be able to

focus on addressing your application's functionality needs with minimal (or possibly no) thought of configuring Spring itself. In fact, one of the main things that Spring Boot does is to get Spring out of your way so you can get stuff done.

Throughout the chapters in this book, we'll explore various facets of Spring Boot development. But first, let's take a high-level look at what Spring Boot has to offer.

1.1 *Spring rebooted*

Spring started as a lightweight alternative to Java Enterprise Edition (JEE, or J2EE as it was known at the time). Rather than develop components as heavyweight Enterprise JavaBeans (EJBs), Spring offered a simpler approach to enterprise Java development, utilizing dependency injection and aspect-oriented programming to achieve the capabilities of EJB with plain old Java objects (POJOs).

But while Spring was lightweight in terms of component code, it was heavyweight in terms of configuration. Initially, Spring was configured with XML (and lots of it). Spring 2.5 introduced annotation-based component-scanning, which eliminated a great deal of explicit XML configuration for an application's own components. And Spring 3.0 introduced a Java-based configuration as a type-safe and refactorable option to XML.

Even so, there was no escape from configuration. Enabling certain Spring features such as transaction management and Spring MVC required explicit configuration, either in XML or Java. Enabling third-party library features such as Thymeleaf-based web views required explicit configuration. Configuring servlets and filters (such as Spring's `DispatcherServlet`) required explicit configuration in `web.xml` or in a servlet initializer. Component-scanning reduced configuration and Java configuration made it less awkward, but Spring still required a lot of configuration.

All of that configuration represents development friction. Any time spent writing configuration is time spent not writing application logic. The mental shift required to think about configuring a Spring feature distracts from solving the business problem. Like any framework, Spring does a lot for you, but it demands that you do a lot for it in return.

Moreover, project dependency management is a thankless task. Deciding what libraries need to be part of the project build is tricky enough. But it's even more challenging to know which versions of those libraries will play well with others.

As important as it is, dependency management is another form of friction. When you're adding dependencies to your build, you're not writing application code. Any incompatibilities that come from selecting the wrong versions of those dependencies can be a real productivity killer.

Spring Boot has changed all of that.

1.1.1 *Taking a fresh look at Spring*

Suppose you're given the task of developing a very simple Hello World web application with Spring. What would you need to do? I can think of a handful of things you'd need at a bare minimum:

- A project structure, complete with a Maven or Gradle build file including required dependencies. At the very least, you'll need Spring MVC and the Servlet API expressed as dependencies.
- A `web.xml` file (or a `WebApplicationInitializer` implementation) that declares Spring's `DispatcherServlet`.
- A Spring configuration that enables Spring MVC.
- A controller class that will respond to HTTP requests with "Hello World".
- A web application server, such as Tomcat, to deploy the application to.

What's most striking about this list is that only one item is specific to developing the Hello World functionality: the controller. The rest of it is generic boilerplate that you'd need for any web application developed with Spring. But if all Spring web applications need it, why should you have to provide it?

Suppose for a moment that the controller is all you need. As it turns out, the Groovy-based controller class shown in listing 1.1 is a complete (even if simple) Spring application.

Listing 1.1 A complete Groovy-based Spring application

```
@RestController
class HelloController {

    @RequestMapping("/")
    def hello() {
        return "Hello World"
    }
}
```

There's no configuration. No `web.xml`. No build specification. Not even an application server. This is the entire application. Spring Boot will handle the logistics of executing the application. You only need to bring the application code.

Assuming that you have Spring Boot's command-line interface (CLI) installed, you can run `HelloController` at the command line like this:

```
$ spring run HelloController.groovy
```

You may have also noticed that it wasn't even necessary to compile the code. The Spring Boot CLI was able to run it from its uncompiled form.

I chose to write this example controller in Groovy because the simplicity of the Groovy language presents well alongside the simplicity of Spring Boot. But Spring Boot doesn't require that you use Groovy. In fact, much of the code we'll write in this book will be in Java. But there'll be some Groovy here and there, where appropriate.

Feel free to look ahead to section 1.21 to see how to install the Spring Boot CLI, so that you can try out this little web application. But for now, we'll look at the key pieces of Spring Boot to see how it changes Spring application development.

1.1.2 Examining Spring Boot essentials

Spring Boot brings a great deal of magic to Spring application development. But there are four core tricks that it performs:

- *Automatic configuration*—Spring Boot can automatically provide configuration for application functionality common to many Spring applications.
- *Starter dependencies*—You tell Spring Boot what kind of functionality you need, and it will ensure that the libraries needed are added to the build.
- *The command-line interface*—This optional feature of Spring Boot lets you write complete applications with just application code, but no need for a traditional project build.
- *The Actuator*—Gives you insight into what's going on inside of a running Spring Boot application.

Each of these features serves to simplify Spring application development in its own way. We'll look at how to employ them to their fullest throughout this book. But for now, let's take a quick look at what each offers.

AUTO-CONFIGURATION

In any given Spring application's source code, you'll find either Java configuration or XML configuration (or both) that enables certain supporting features and functionality for the application. For example, if you've ever written an application that accesses a relational database with JDBC, you've probably configured Spring's `JdbcTemplate` as a bean in the Spring application context. I'll bet the configuration looked a lot like this:

```
@Bean
public JdbcTemplate jdbcTemplate(DataSource dataSource) {
    return new JdbcTemplate(dataSource);
}
```

This very simple bean declaration creates an instance of `JdbcTemplate`, injecting it with its one dependency, a `DataSource`. Of course, that means that you'll also need to configure a `DataSource` bean so that the dependency will be met. To complete this configuration scenario, suppose that you were to configure an embedded H2 database as the `DataSource` bean:

```
@Bean
public DataSource dataSource() {
    return new EmbeddedDatabaseBuilder()
        .setType(EmbeddedDatabaseType.H2)
        .addScripts('schema.sql', 'data.sql')
        .build();
}
```

This bean configuration method creates an embedded database, specifying two SQL scripts to execute on the embedded database. The `build()` method returns a `DataSource` that references the embedded database.

Neither of these two bean configuration methods is terribly complex or lengthy. But they represent just a fraction of the configuration in a typical Spring application. Moreover, there are countless Spring applications that will have these exact same methods. Any application that needs an embedded database and a `JdbcTemplate` will need those methods. In short, it's boilerplate configuration.

If it's so common, then why should you have to write it?

Spring Boot can automatically configure these common configuration scenarios. If Spring Boot detects that you have the H2 database library in your application's classpath, it will automatically configure an embedded H2 database. If `JdbcTemplate` is in the classpath, then it will also configure a `JdbcTemplate` bean for you. There's no need for you to worry about configuring those beans. They'll be configured for you, ready to inject into any of the beans you write.

There's a lot more to Spring Boot auto-configuration than embedded databases and `JdbcTemplate`. There are several dozen ways that Spring Boot can take the burden of configuration off your hands, including auto-configuration for the Java Persistence API (JPA), Thymeleaf templates, security, and Spring MVC. We'll dive into auto-configuration starting in chapter 2.

STARTER DEPENDENCIES

It can be challenging to add dependencies to a project's build. What library do you need? What are its group and artifact? Which version do you need? Will that version play well with other dependencies in the same project?

Spring Boot offers help with project dependency management by way of starter dependencies. Starter dependencies are really just special Maven (and Gradle) dependencies that take advantage of transitive dependency resolution to aggregate commonly used libraries under a handful of feature-defined dependencies.

For example, suppose that you're going to build a REST API with Spring MVC that works with JSON resource representations. Additionally, you want to apply declarative validation per the JSR-303 specification and serve the application using an embedded Tomcat server. To accomplish all of this, you'll need (at minimum) the following eight dependencies in your Maven or Gradle build:

- `org.springframework:spring-core`
- `org.springframework:spring-web`
- `org.springframework:spring-webmvc`
- `com.fasterxml.jackson.core:jackson-databind`
- `org.hibernate:hibernate-validator`
- `org.apache.tomcat.embed:tomcat-embed-core`
- `org.apache.tomcat.embed:tomcat-embed-el`
- `org.apache.tomcat.embed:tomcat-embed-logging-juli`

On the other hand, if you were to take advantage of Spring Boot starter dependencies, you could simply add the Spring Boot "web" starter (`org.springframework.boot:spring-boot-starter-web`) as a build dependency. This single dependency

will transitively pull in all of those other dependencies so you don't have to ask for them all.

But there's something more subtle about starter dependencies than simply reducing build dependency count. Notice that by adding the "web" starter to your build, you're specifying a type of functionality that your application needs. Your app is a web application, so you add the "web" starter. Likewise, if your application will use JPA persistence, then you can add the "jpa" starter. If it needs security, you can add the "security" starter. In short, you no longer need to think about what libraries you'll need to support certain functionality; you simply ask for that functionality by way of the pertinent starter dependency.

Also note that Spring Boot's starter dependencies free you from worrying about which versions of these libraries you need. The versions of the libraries that the starters pull in have been tested together so that you can be confident that there will be no incompatibilities between them.

Along with auto-configuration, we'll begin using starter dependencies right away, starting in chapter 2.

THE COMMAND-LINE INTERFACE (CLI)

In addition to auto-configuration and starter dependencies, Spring Boot also offers an intriguing new way to quickly write Spring applications. As you saw earlier in section 1.1, the Spring Boot CLI makes it possible to write applications by doing more than writing the application code.

Spring Boot's CLI leverages starter dependencies and auto-configuration to let you focus on writing code. Not only that, did you notice that there are no `import` lines in listing 1.1? How did the CLI know what packages `RequestMapping` and `RestController` come from? For that matter, how did those classes end up in the classpath?

The short answer is that the CLI detected that those types are being used, and it knows which starter dependencies to add to the classpath to make it work. Once those dependencies are in the classpath, a series of auto-configuration kicks in and ensures that `DispatcherServlet` and Spring MVC are enabled so that the controller can respond to HTTP requests.

Spring Boot's CLI is an optional piece of Spring Boot's power. Although it provides tremendous power and simplicity for Spring development, it also introduces a rather unconventional development model. If this development model is too extreme for your taste, then no problem. You can still take advantage of everything else that Spring Boot has to offer even if you don't use the CLI. But if you like what the CLI provides, you'll definitely want to look at chapter 5 where we'll dig deeper into Spring Boot's CLI.

THE ACTUATOR

The final piece of the Spring Boot puzzle is the Actuator. Where the other parts of Spring Boot simplify Spring development, the Actuator instead offers the ability to inspect the internals of your application at runtime. With the Actuator installed, you can inspect the inner workings of your application, including details such as

- What beans have been configured in the Spring application context
- What decisions were made by Spring Boot's auto-configuration
- What environment variables, system properties, configuration properties, and command-line arguments are available to your application
- The current state of the threads in and supporting your application
- A trace of recent HTTP requests handled by your application
- Various metrics pertaining to memory usage, garbage collection, web requests, and data source usage

The Actuator exposes this information in two ways: via web endpoints or via a shell interface. In the latter case, you can actually open a secure shell (SSH) into your application and issue commands to inspect your application as it runs.

We'll explore the Actuator's capabilities in detail when we get to chapter 7.

1.1.3 What Spring Boot isn't

Because of the amazing things Spring Boot does, there has been a lot of talk about Spring Boot in the past year or so. Depending on what you've heard or read about Spring Boot before reading this book, you may have a few misconceptions about Spring Boot that should be cleared up before continuing.

First, Spring Boot is not an application server. This misconception stems from the fact that it's possible to create web applications as self-executable JAR files that can be run at the command line without deploying applications to a conventional Java application server. Spring Boot accomplishes this by embedding a servlet container (Tomcat, Jetty, or Undertow) within the application. But it's the embedded servlet container that provides application server functionality, not Spring Boot itself.

Similarly, Spring Boot doesn't implement any enterprise Java specifications such as JPA or JMS. It does support several enterprise Java specifications, but it does so by automatically configuring beans in Spring that support those features. For instance, Spring Boot doesn't implement JPA, but it does support JPA by auto-configuring the appropriate beans for a JPA implementation (such as Hibernate).

Finally, Spring Boot doesn't employ any form of code generation to accomplish its magic. Instead, it leverages conditional configuration features from Spring 4, along with transitive dependency resolution offered by Maven and Gradle, to automatically configure beans in the Spring application context.

In short, at its heart, Spring Boot is just Spring. Inside, Spring Boot is doing the same kind of bean configuration in Spring that you might do on your own if Spring Boot didn't exist. Thankfully, because Spring Boot does exist, you're freed from dealing with explicit boilerplate configuration and are able to focus on the logic that makes your application unique.

By now you should have a general idea of what Spring Boot brings to the table. It's just about time for you to build your first application with Spring Boot. First things first, though. Let's see how you can take your first steps with Spring Boot.

1.2 Getting started with Spring Boot

Ultimately, a Spring Boot project is just a regular Spring project that happens to leverage Spring Boot starters and auto-configuration. Therefore, any technique or tool you may already be familiar with for creating a Spring project from scratch will apply to a Spring Boot project. There are, however, a few convenient options available for kick-starting your project with Spring Boot.

The quickest way to get started with Spring Boot is to install the Spring Boot CLI so that you can start writing code, such as that in listing 1.1, that runs via the CLI.

1.2.1 Installing the Spring Boot CLI

As we discussed earlier, the Spring Boot CLI offers an interesting, albeit unconventional, approach to developing Spring applications. We'll dive into the specifics of what the CLI offers in chapter 5. But for now let's look at how to install the Spring Boot CLI so that you can run the code we looked at in listing 1.1.

There are several ways to install the Spring Boot CLI:

- From a downloaded distribution
- Using the Groovy Environment Manager
- With OS X Homebrew
- As a port using MacPorts

We'll look at each installation option. In addition, we'll also see how to install support for Spring Boot CLI command completion, which comes in handy if you're using the CLI on BASH or zsh shells (sorry, Windows users). Let's first look at how you can install the Spring Boot CLI manually from a distribution.

MANUALLY INSTALLING THE SPRING BOOT CLI

Perhaps the most straightforward way to install the Spring Boot CLI is to download it, unzip it, and add its bin directory to your path. You can download the distribution archive from either of these locations:

- <http://repo.spring.io/release/org/springframework/boot/spring-boot-cli/1.3.0.RELEASE/spring-boot-cli-1.3.0.RELEASE-bin.zip>
- <http://repo.spring.io/release/org/springframework/boot/spring-boot-cli/1.3.0.RELEASE/spring-boot-cli-1.3.0.RELEASE-bin.tar.gz>

Once you've downloaded the distribution, unpack it somewhere in your filesystem. Inside of the unpacked archive, you'll find a bin directory that contains a `spring.bat` script (for Windows) and a `spring` script for Unix. Add this bin directory to your system path and you're ready to use the Spring Boot CLI.

SYMBOLICALLY LINKING TO SPRING BOOT If you're using the Spring Boot CLI on a Unix machine, it may be helpful to create a symbolic link to the unpacked archive and add the symbolic link to your path instead of the actual directory. This will make it easy to upgrade to a newer version of Spring Boot later (or even to flip between versions) by simply reassigning the symbolic link to the directory of the new version.

You can kick the tires a little on the installation by verifying the version of the CLI that was installed:

```
$ spring --version
```

If everything is working, you'll be shown the version of the Spring Boot CLI that was installed.

Even though this is the manual installation, it's an easy option that doesn't require you to have anything additional installed. If you're a Windows user, it's also the only choice available to you. But if you're on a Unix machine and are looking for something a little more automated, then maybe the Software Development Kit Manager can help.

INSTALLING WITH THE SOFTWARE DEVELOPMENT KIT MANAGER

The Software Development Kit Manager (SDKMAN; formerly known as GVM) can be used to install and manage multiple versions of Spring Boot CLI installations. In order to use SDKMAN, you'll need to get and install the SDKMAN tool from <http://sdkman.io>. The easiest way to install SDKMAN is at the command line:

```
$ curl -s get.sdkman.io | bash
```

Follow the instructions given in the output to complete the SDKMAN installation. For my machine, I had to perform the following command at the command line:

```
$ source "/Users/habuma/.sdkman/bin/sdkman-init.sh"
```

Note that this command will be different for different users. In my case, my home directory is at /Users/habuma, so that's the root of the shell script's path. You'll want to adjust accordingly to fit your situation.

Once SDKMAN is installed, you can install Spring Boot's CLI like this:

```
$ sdk install springboot
$ spring --version
```

Assuming all goes well, you'll be shown the current version of Spring Boot.

If you want to upgrade to a newer version of Spring Boot CLI, you just need to install it and start using it. To find out which versions of Spring Boot CLI are available, use SDKMAN's list command:

```
$ sdk list springboot
```

The list command shows all available versions, including which versions are installed and which is currently in use. From this list you can choose to install a version and then use it. For example, to install Spring Boot CLI version 1.3.0.RELEASE, you'd use the install command, specifying the version:

```
$ sdk install springboot 1.3.0.RELEASE
```

This will install the new version and ask if you'd like to make it the default version. If you choose not to make it the default version or if you wish to switch to a different version, you can use the `use` command:

```
$ sdk use springboot 1.3.0.RELEASE
```

If you'd like that version to be the default for all shells, use the `default` command:

```
$ sdk default springboot 1.3.0.RELEASE
```

The nice thing about using SDKMAN to manage your Spring Boot CLI installation is that it allows you to easily switch between different versions of Spring Boot. This will enable you to try out snapshot, milestone, and release candidate builds before they're formally released, but still switch back to a stable release for other work.

INSTALLING WITH HOMEBREW

If you'll be developing on an OS X machine, you have the option of using Homebrew to install the Spring Boot CLI. Homebrew is a package manager for OS X that is used to install many different applications and tools. The easiest way to install Homebrew is by running the installation Ruby script:

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

You can read more about Homebrew (and find other installation options) at <http://brew.sh>.

In order to install the Spring Boot CLI using Homebrew, you'll need to "tap" Pivotal's tap:¹

```
$ brew tap pivotal/tap
```

Now that Homebrew is tapping Pivotal's tap, you can install the Spring Boot CLI like this:

```
$ brew install springboot
```

Homebrew will install the Spring Boot CLI to `/usr/local/bin`, and it's ready to go. You can verify the installation by checking the version that was installed:

```
$ spring --version
```

It should respond by showing you the version of Spring Boot that was installed. You can also try running the code in listing 1.1.

INSTALLING WITH MACPORTS

Another Spring Boot CLI installation option for OS X users is to use MacPorts, another popular installer for Mac OS X. In order to use MacPorts to install the Spring Boot

¹ Tapping is a way to add additional repositories to those that Homebrew works from. Pivotal, the company behind Spring and Spring Boot, has made the Spring Boot CLI available through its tap.

CLI, you must first install MacPorts, which itself requires that you have Xcode installed. Furthermore, the steps for installing MacPorts vary depending on which version of OS X you're using. Therefore, I refer you to <https://www.macports.org/install.php> for instructions on installing MacPorts.

Once you have MacPorts installed, you can install the Spring Boot CLI at the command line like this:

```
$ sudo port install spring-boot-cli
```

MacPorts will install the Spring Boot CLI to `/opt/local/share/java/spring-boot-cli` and put a symbolic link to the binary in `/opt/local/bin`, which should already be in your system path from installing MacPorts. You can verify the installation by checking the version that was installed:

```
$ spring --version
```

It should respond by showing you the version of Spring Boot that was installed. You can also try running the code in listing 1.1.

ENABLING COMMAND-LINE COMPLETION

Spring Boot's CLI offers a handful of commands for running, packaging, and testing your CLI-based application. Moreover, each of those commands has several options. It can be difficult to remember all that the CLI offers. Command-line completion can help you recall how to use the Spring Boot CLI.

If you've installed the Spring Boot CLI with Homebrew, you already have command-line completion installed. But if you installed Spring Boot manually or with SDKMAN, you'll need to source the scripts or install the completion scripts manually. (Command-line completion isn't an option if you've installed the Spring Boot CLI via MacPorts.)

The completion scripts are found in the Spring Boot CLI installation directory under the `shell-completion` subdirectory. There are two different scripts, one for BASH and one for `zsh`. To source the completion script for BASH, you can enter the following at the command line (assuming a SDKMAN installation):

```
$ . ~/.sdkman/springboot/current/shell-completion/bash/spring
```

This will give you Spring Boot CLI completion for the current shell, but you'll have to source this script again each time you start a new shell to keep that feature. Optionally, you can copy the script to your personal or system script directory. The location of the script directory varies for different Unix installations, so consult your system documentation (or Google) for details.

With command completion enabled, you should be able to type `spring` at the command line and then hit the Tab key to be offered options for what to type next. Once you've chosen a command, type `--` (double-hyphen) and then hit Tab again to be shown a list of options for that command.

If you're developing on Windows or aren't using BASH or zsh, you can't use these command-line completion scripts. Even so, you can get command completion if you run the Spring Boot CLI shell:

```
$ spring shell
```

Unlike the command-completion scripts for BASH and zsh (which operate within the BASH/zsh shell), the Spring Boot CLI shell opens a new Spring Boot-specific shell. From this shell, you can execute any of the CLI's commands and get command completion with the Tab key.

The Spring Boot CLI offers an easy way to get started with Spring Boot and to prototype simple applications. As we'll discuss later in chapter 8, it can also be used for production-ready applications, given the right production runtime environment.

Even so, Spring Boot CLI's process is rather unconventional in contrast to how most Java projects are developed. Typically, Java projects use tools like Gradle or Maven to build WAR files that are deployed to an application server. If the CLI model feels a little uncomfortable, you can still take advantage of most of the features of Spring Boot in the context of a traditionally built Java project.² And the Spring Initializr can help you get started.

1.2.2 *Initializing a Spring Boot project with Spring Initializr*

Sometimes the hardest part of a project is getting started. You need to set up a directory structure for various project artifacts, create a build file, and populate the build file with dependencies. The Spring Boot CLI removes much of this setup work, but if you favor a more traditional Java project structure, you'll want to look at the Spring Initializr.

The Spring Initializr is ultimately a web application that can generate a Spring Boot project structure for you. It doesn't generate any application code, but it will give you a basic project structure and either a Maven or a Gradle build specification to build your code with. All you need to do is write the application code.

Spring Initializr can be used in several ways:

- Through a web-based interface
- Via Spring Tool Suite
- Via IntelliJ IDEA
- Using the Spring Boot CLI

We'll look at how to use each of these interfaces to the Initializr, starting with the web-based interface.

² You'll only be giving up features that require the flexibility of the Groovy language, such as automatic dependency and import resolution.

USING SPRING INITIALIZR'S WEB INTERFACE

The most straightforward way to use the Spring Initializr is to point your web browser to <http://start.spring.io>. You should see a form similar to the one in figure 1.1.

The first two things that the form asks is whether you want to build your project with Maven or Gradle and which version of Spring Boot to use. It defaults to a Maven project using the latest release (non-milestone, non-snapshot) version of Spring Boot, but you're welcome to choose a different one.

On the left side of the form, you're asked to specify some project metadata. At minimum, you must provide the project's group and artifact. But if you click the "Switch to the full version" link, you can specify additional metadata such as version and base package name. This metadata is used to populate the generated Maven pom.xml file (or Gradle build.gradle file).

The screenshot shows the Spring Initializr web application in a browser window. The address bar shows start.spring.io. The page title is "Spring Initializr". The main heading is "SPRING INITIALIZR bootstrap your application now". Below this, there's a form to "Generate a" project. The "Project Type" is set to "Maven Project" and the "Spring Boot Version" is set to "1.3.0". The "Project Metadata" section has fields for "Group" (com.example) and "Artifact" (demo). A green "Generate Project" button is visible. The "Dependencies" section has a search bar with "web" entered. A dropdown menu shows several dependency options: "Web" (Full-stack web development with Tomcat and Spring MVC), "Rest Repositories" (Exposing Spring Data repositories over REST via spring-data-rest-webmvc), "Websocket" (Websocket development with SockJS and STOMP), "WS" (Contract-first SOAP service development with Spring Web Services), and "Jersey (JAX-RS)" (the Jersey RESTful Web Services framework). A link "Switch to the full version." is also present.

Figure 1.1 Spring Initializr is a web application that generates empty Spring projects as starting points for development.

On the right side of the form, you're asked to specify project dependencies. The easiest way to do that is to type the name of a dependency in the text box. As you type, a list of matching dependencies will appear. Select the one(s) you want and it will be added to the project. If you don't see what you're looking for, click the "Switch to the full version" link to get a complete list of available dependencies.

If you've glanced at appendix B, then you'll recognize that the dependencies offered correspond to Spring Boot starter dependencies. In fact, by selecting any of these dependencies, you're telling the Initializr to add the starters as dependencies to the project's build file. (We'll talk more about Spring Boot starters in chapter 2.)

Once you've filled in the form and made your dependency selections, click the Generate Project button to have Spring Initializr generate a project for you. The project it generates will be presented to you as a zip file (whose name is determined by the value in the Artifact field) that is downloaded by your browser. The contents of the zip file will vary slightly, depending on the choices you made before clicking Generate Project. In any event, the zip file will contain a bare-bones project to get you started developing an application with Spring Boot.

For example, suppose that you were to specify the following to Spring Initializr:

- Artifact: myapp
- Package Name: myapp
- Type: Gradle Project
- Dependencies: Web and JPA

After clicking Generate Project, you'd be given a zip file named myapp.zip. After unzipping it, you'd have a project structure similar to what's shown in figure 1.2.

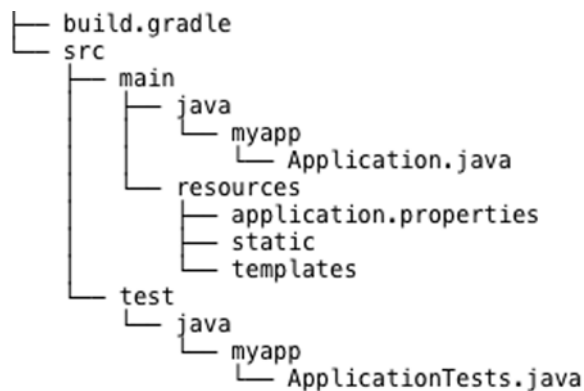


Figure 1.2 Initializr-created projects provide a minimal foundation on which to build Spring Boot applications.

As you can see, there's very little code in this project. Aside from a couple of empty directories, it also includes the following:

- `build.gradle`—A Gradle build specification. Had you chosen a Maven project, this would be replaced with `pom.xml`.
- `Application.java`—A class with a `main()` method to bootstrap the application.
- `ApplicationTests.java`—An empty JUnit test class instrumented to load a Spring application context using Spring Boot auto-configuration.
- `application.properties`—An empty properties file for you to add configuration properties to as you see fit.

Even the empty directories have significance in a Spring Boot application. The `static` directory is where you can put any static content (JavaScript, stylesheets, images, and so on) to be served from the web application. And, as you'll see later, you can put templates that render model data in the `templates` directory.

You'll probably import the Initializr-created project into your IDE of choice. But if Spring Tool Suite is your IDE of choice, you can create the project directly in the IDE. Let's have a look at Spring Tool Suite's support for creating Spring Boot projects.

CREATING SPRING BOOT PROJECTS IN SPRING TOOL SUITE

Spring Tool Suite³ has long been a fantastic IDE for developing Spring applications. Since version 3.4.0 it has also been integrated with the Spring Initializr, making it a great way to get started with Spring Boot.

To create a new Spring Boot application in Spring Tool Suite, select the `New > Spring Starter Project` menu item from the `File` menu. When you do, Spring Tool Suite will present you with a dialog box similar to the one shown in figure 1.3.

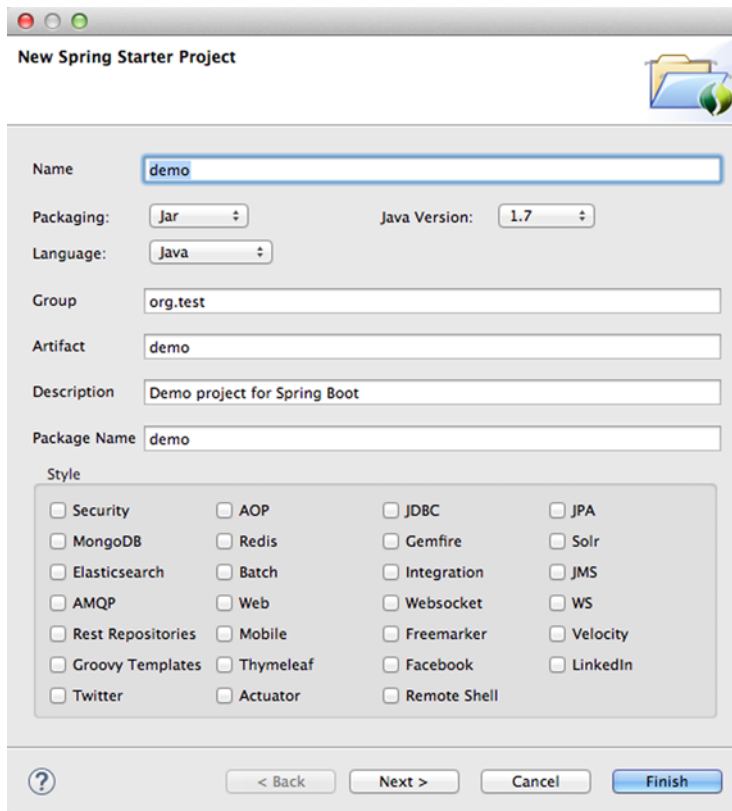
As you can see, this dialog box asks for the same information as the web-based Spring Initializr. In fact, the data you provide here will be fed to Spring Initializr to create a project zip file, just as with the web-based form.

If you'd like to specify where in the filesystem to create the project or whether to add it to a specific working set within the IDE, click the `Next` button. You'll be presented with a second dialog box like the one shown in figure 1.4.

The `Location` field specifies where the project will reside on the filesystem. If you take advantage of Eclipse's working sets to organize your projects, you can have the project added to a specific working set by checking the `Add Project to Working Sets` check box and selecting a working set.

The `Site Info` section simply describes the URL that will be used to contact the Initializr. For the most part, you can ignore this section. If, however, you were to deploy your own Initializr server (by cloning the code at <https://github.com/spring-io/initializr>), you could plug in the base URL of your Initializr here.

³ Spring Tool Suite is a distribution of the Eclipse IDE that is outfitted with several features to aid with Spring development. You can download Spring Tool Suite from <http://spring.io/tools/sts>.



New Spring Starter Project

Name:

Packaging: Java Version:

Language:

Group:

Artifact:

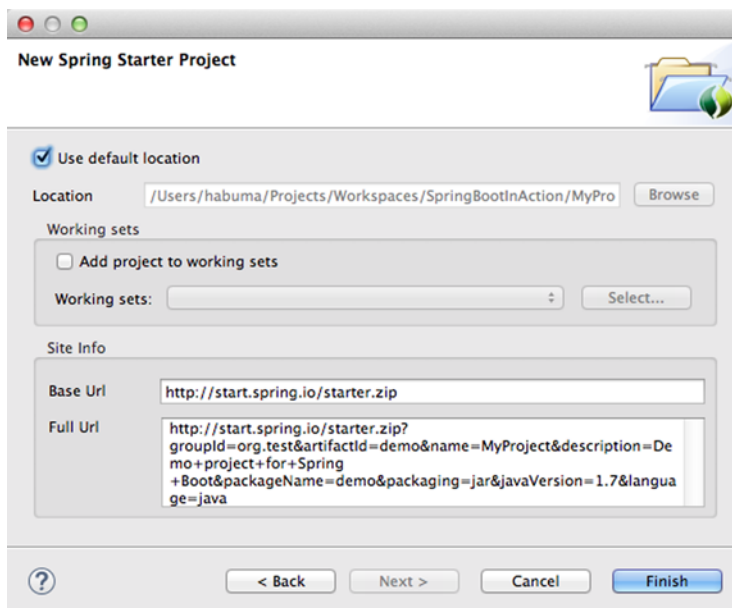
Description:

Package Name:

Style

<input type="checkbox"/> Security	<input type="checkbox"/> AOP	<input type="checkbox"/> JDBC	<input type="checkbox"/> JPA
<input type="checkbox"/> MongoDB	<input type="checkbox"/> Redis	<input type="checkbox"/> Gemfire	<input type="checkbox"/> Solr
<input type="checkbox"/> Elasticsearch	<input type="checkbox"/> Batch	<input type="checkbox"/> Integration	<input type="checkbox"/> JMS
<input type="checkbox"/> AMQP	<input type="checkbox"/> Web	<input type="checkbox"/> Websocket	<input type="checkbox"/> WS
<input type="checkbox"/> Rest Repositories	<input type="checkbox"/> Mobile	<input type="checkbox"/> Freemarker	<input type="checkbox"/> Velocity
<input type="checkbox"/> Groovy Templates	<input type="checkbox"/> Thymeleaf	<input type="checkbox"/> Facebook	<input type="checkbox"/> LinkedIn
<input type="checkbox"/> Twitter	<input type="checkbox"/> Actuator	<input type="checkbox"/> Remote Shell	

Figure 1.3 Spring Tool Suite integrates with Spring Initializr to create and directly import Spring Boot projects into the IDE.



New Spring Starter Project

☒ Use default location

Location:

Working sets

☐ Add project to working sets

Working sets:

Site Info

Base Url:

Full Url:

Figure 1.4 The second page of the Spring Starter Project dialog box offers you a chance to specify where the project is created.

Clicking the Finish button kicks off the project generation and import process. It's important to understand that Spring Tool Suite's Spring Starter Project dialog box delegates to the Spring Initializr at <http://start.spring.io> to produce the project. You must be connected to the internet in order for it to work.

Once the project has been imported into your workspace, you're ready to start developing your application. As you develop the application, you'll find that Spring Tool Suite has a few more Spring Boot-specific tricks up its sleeves. For instance, you can run your application with an embedded server by selecting Run As > Spring Boot Application from the Run menu.

It's important to understand that Spring Tool Suite coordinates with the Initializr via a REST API. Therefore, it will only work if it can connect to the Initializr. If your development machine is offline or Initializr is blocked by a firewall, then using the Spring Start Project wizard in Spring Tool Suite will not work.

CREATING SPRING BOOT PROJECTS IN INTELLIJ IDEA

IntelliJ IDEA is a very popular IDE and, as of IntelliJ IDEA 14.1, it now supports Spring Boot!⁴

To get started on a new Spring Boot application in IntelliJ IDEA, select New > Project from the File menu. You'll be presented with the first of a handful of screens (shown in figure 1.5) that ask questions similar to those asked by the Initializr web application and Spring Tool Suite.

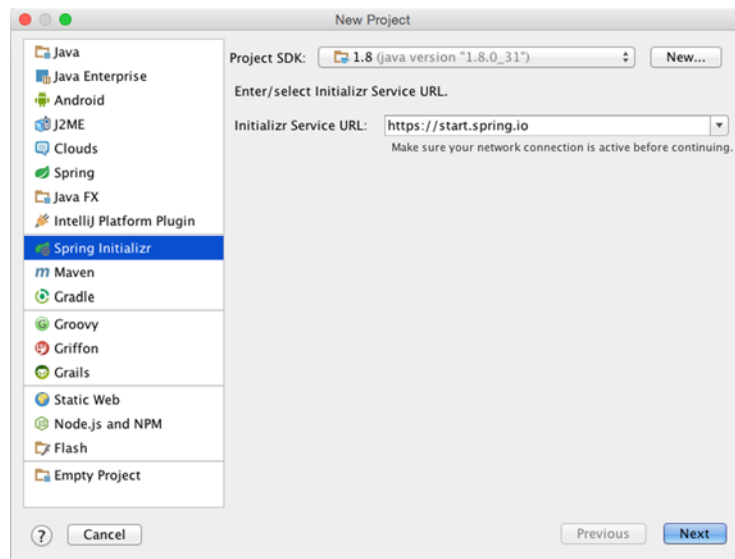
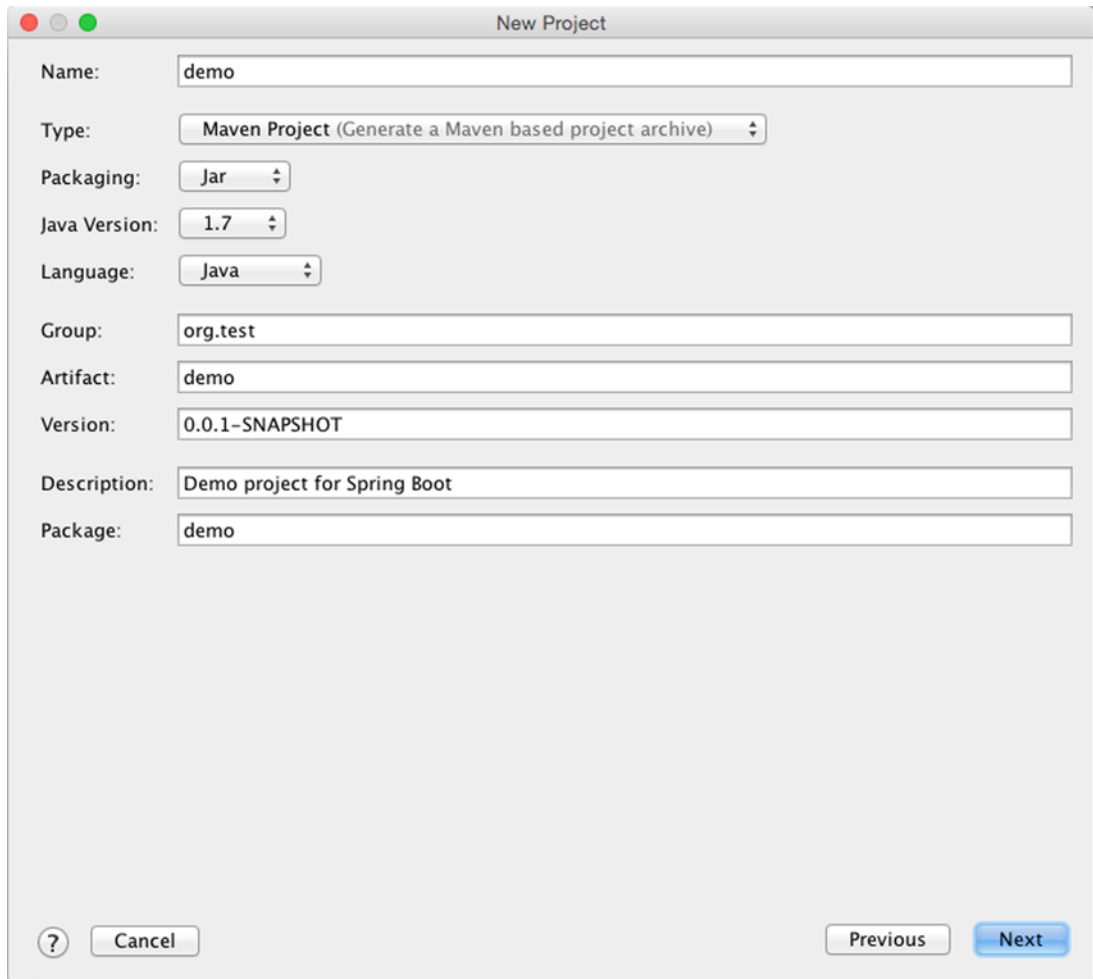


Figure 1.5 The first screen in IntelliJ IDEA's Spring Boot initialization wizard

⁴ You can get IntelliJ IDEA at <https://www.jetbrains.com/idea/>. IntelliJ IDEA is a commercial IDE, meaning that you may have to pay for it. You can, however, download a trial of it, and it's freely available for use on open source projects.

The image shows the 'New Project' dialog box in IntelliJ IDEA. The title bar says 'New Project'. The dialog contains several fields and dropdown menus for configuring a new project. The fields are: Name (demo), Type (Maven Project (Generate a Maven based project archive)), Packaging (Jar), Java Version (1.7), Language (Java), Group (org.test), Artifact (demo), Version (0.0.1-SNAPSHOT), Description (Demo project for Spring Boot), and Package (demo). At the bottom, there are three buttons: a help button (question mark icon), a 'Cancel' button, and a 'Next' button (highlighted in blue). There is also a 'Previous' button, which is disabled.

Name: demo

Type: Maven Project (Generate a Maven based project archive)

Packaging: Jar

Java Version: 1.7

Language: Java

Group: org.test

Artifact: demo

Version: 0.0.1-SNAPSHOT

Description: Demo project for Spring Boot

Package: demo

? Cancel Previous Next

Figure 1.6 Specifying project information in IntelliJ IDEA's Spring Boot initialization wizard

On the initial screen, select Spring Initializr from the project choices on the left. You'll then be prompted to select a Project SDK (essentially, which Java SDK you want to use for the project) and the location of the Initializr web service. Unless you're running your own instance of the Initializr, you'll probably just click the Next button here without making any changes. That will take you to the screen shown in figure 1.6.

The second screen in IntelliJ IDEA's Spring Boot initialization wizard asks some basic questions about the project, such as the project's name, Maven group and artifact, Java version, and whether you want to build it with Maven or Gradle. Once you've described your project, clicking the Next button takes you to the third screen, shown in figure 1.7.

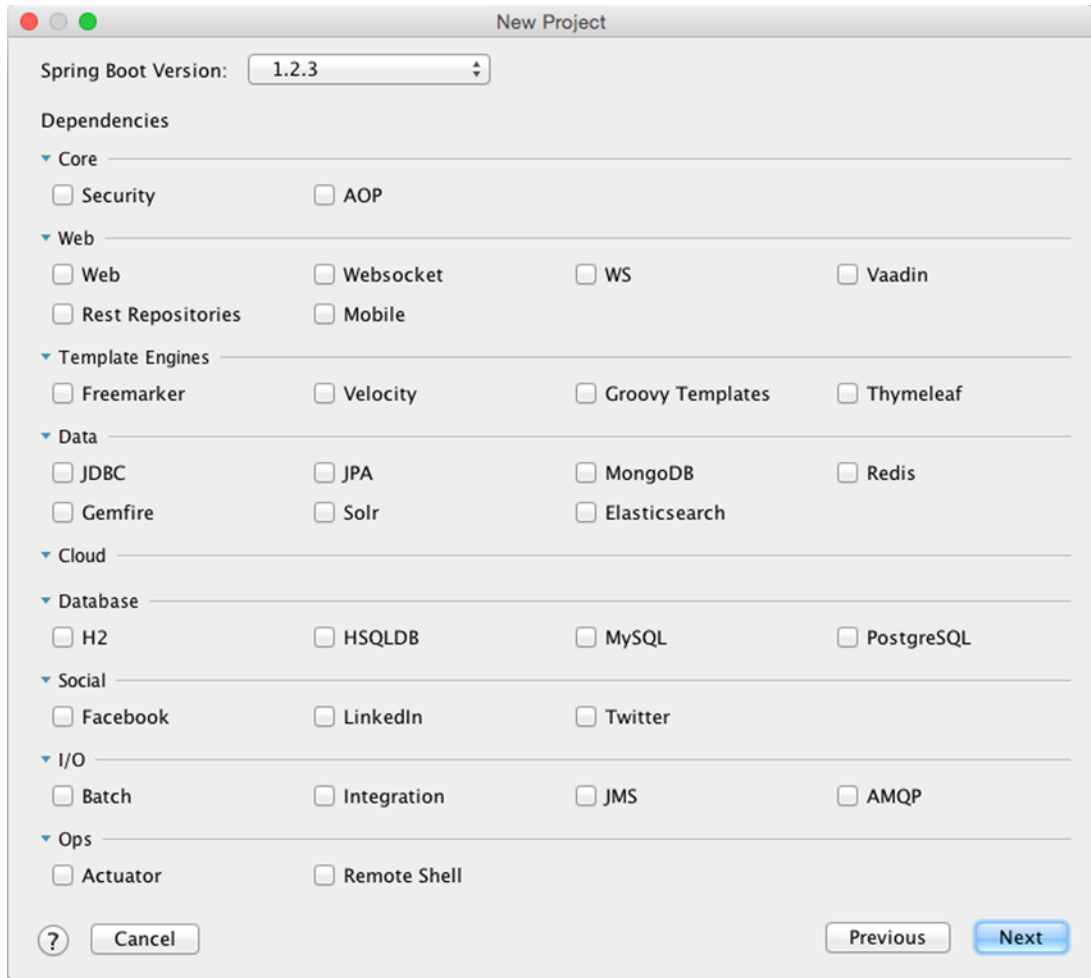


Figure 1.7 Selecting project dependencies in IntelliJ IDEA's Spring Boot initialization wizard

Where the second screen asked you about general project information, the third screen starts by asking you what kind of dependencies you'll need in the project. As before, the check boxes shown on this screen correspond to Spring Boot starter dependencies. After you've made your selections, click Next to be taken to the final screen in the wizard, shown in figure 1.8.

This last screen simply wants you to name the project and tell IntelliJ IDEA where to create it. When you're ready, click the Finish button and you'll have a bare-bones Spring Boot project ready for you in the IDE.

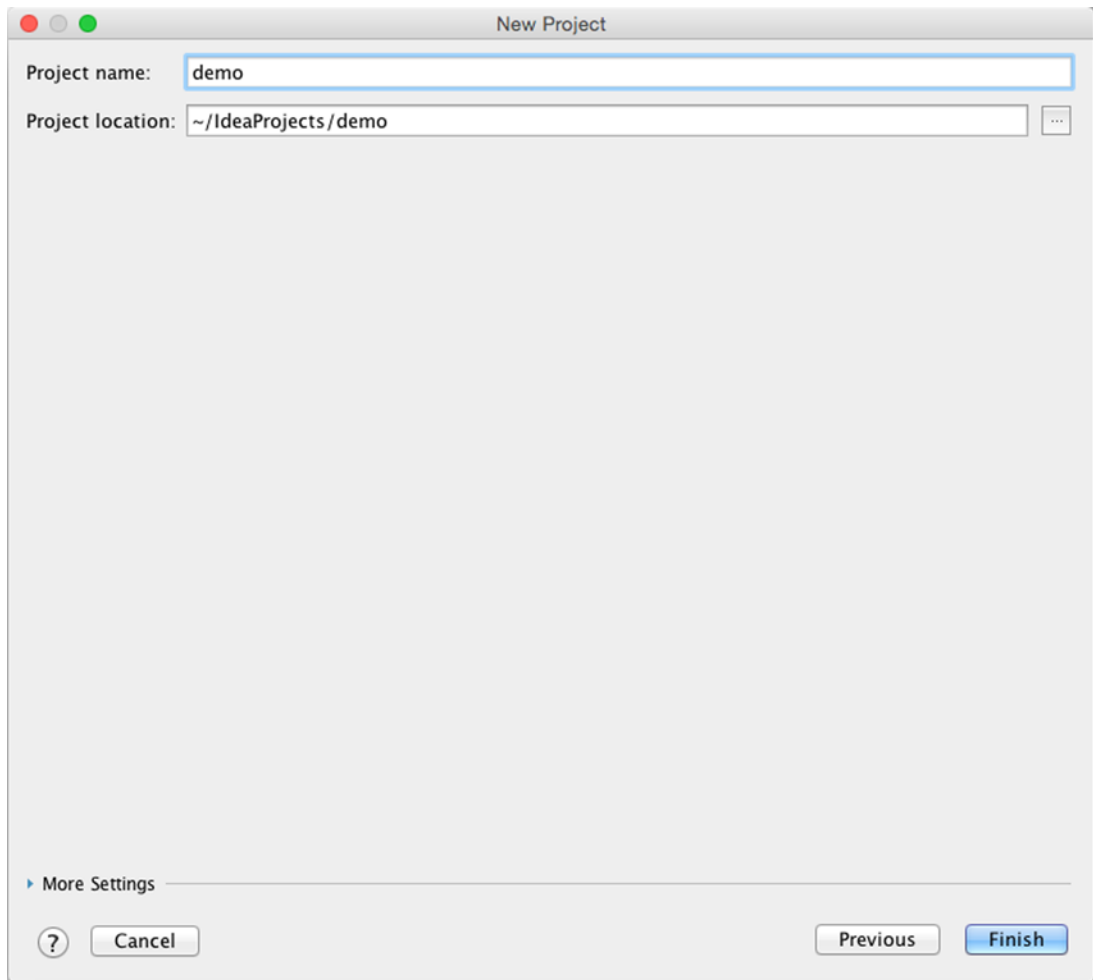


Figure 1.8 The final screen in IntelliJ IDEA's Spring Boot initialization wizard

USING THE INITIALIZR FROM THE SPRING BOOT CLI

As you saw earlier, the Spring Boot CLI is a great way to develop Spring applications by just writing code. However, the Spring Boot CLI also has a few commands that can help you use the Initializr to kick-start development on a more traditional Java project.

The Spring Boot CLI includes an `init` command that acts as a client interface to the Initializr. The simplest use of the `init` command is to create a baseline Spring Boot project:

```
$ spring init
```

After contacting the Initializr web application, the `init` command will conclude by downloading a `demo.zip` file. If you unzip this project, you'll find a typical project

structure with a Maven pom.xml build specification. The Maven build specification is minimal, with only baseline starter dependencies for Spring Boot and testing. You'll probably want a little more than that.

Let's say you want to start out by building a web application that uses JPA for data persistence and that's secured with Spring Security. You can specify those initial dependencies with either `--dependencies` or `-d`:

```
$ spring init -dweb,jpa,security
```

This will give you a demo.zip containing the same project structure as before, but with Spring Boot's web, JPA, and security starters expressed as dependencies in pom.xml. Note that it's important to not type a space between `-d` and the dependencies. Failing to do so will result in the ZIP file being downloaded with the name `web, jpa, security`.

Now let's say that you'd rather build this project with Gradle. No problem. Just specify Gradle as the build type with the `--build` parameter:

```
$ spring init -dweb,jpa,security --build gradle
```

By default, the build specification for both Maven and Gradle builds will produce an executable JAR file. If you'd rather produce a WAR file, you can specify so with the `--packaging` or `-p` parameter:

```
$ spring init -dweb,jpa,security --build gradle -p war
```

So far, the ways we've used the `init` command have resulted in a zip file being downloaded. If you'd like for the CLI to crack open that zip file for you, you can specify a directory for the project to be extracted to:

```
$ spring init -dweb,jpa,security --build gradle -p war myapp
```

The last parameter given here indicates that you want the project to be extracted to the `myapp` directory.

Optionally, if you want the CLI to extract the generated project into the current directory, you can use either the `--extract` or the `-x` parameter:

```
$ spring init -dweb,jpa,security --build gradle -p jar -x
```

The `init` command has several other parameters, including parameters for building a Groovy-based project, specifying the Java version to compile with, and selecting a version of Spring Boot to build against. You can discover all of the parameters by using the `help` command:

```
$ spring help init
```

You can also find out what choices are available for those parameters by using the `--list` or `-l` parameter with the `init` command:

```
$ spring init -l
```

You'll notice that although `spring init -l` lists several parameters that are supported by the Initializr, not all of those parameters are directly supported by the Spring Boot CLI's `init` command. For instance, you can't specify the root package name when initializing a project with the CLI; it will default to "demo". `spring help init` can help you discover what parameters are supported by the CLI's `init` command.

Whether you use Initializr's web-based interface, create your projects from Spring Tool Suite, or use the Spring Boot CLI to initialize a project, projects created using the Spring Boot Initializr have a familiar project layout, not unlike other Java projects you may have developed before.

1.3 *Summary*

Spring Boot is an exciting new way to develop Spring applications with minimal friction from the framework itself. Auto-configuration eliminates much of the boilerplate configuration that infests traditional Spring applications. Spring Boot starters enable you to specify build dependencies by what they offer rather than use explicit library names and version. The Spring Boot CLI takes Spring Boot's frictionless development model to a whole new level by enabling quick and easy development with Groovy from the command line. And the Actuator lets you look inside your running application to see what and how Spring Boot has done.

This chapter has given you a quick overview of what Spring Boot has to offer. You're probably itching to get started on writing a real application with Spring Boot. That's exactly what we'll do in the next chapter. With all that Spring Boot does for you, the hardest part will be turning this page to chapter 2.

Developing your first Spring Boot application

This chapter covers

- Working with Spring Boot starters
- Automatic Spring configuration

When's the last time you went to a supermarket or major retail store and actually had to push the door open? Most large stores have automatic doors that sense your presence and open for you. Any door will enable you to enter a building, but automatic doors don't require that you push or pull them open.

Similarly, many public facilities have restrooms with automatic water faucets and towel dispensers. Although not quite as prevalent as automatic supermarket doors, these devices don't ask much of you and instead are happy to dispense water and towels.

And I honestly don't remember the last time I even saw an ice tray, much less filled it with water or cracked it to get ice for a glass of water. My refrigerator/freezer somehow magically always has ice for me and is at the ready to fill a glass for me.

I bet you can think of countless ways that modern life is automated with devices that work for you, not the other way around. With all of this automation

everywhere, you'd think that we'd see more of it in our development tasks. Strangely, that hasn't been so.

Up until recently, creating an application with Spring required you to do a lot of work for the framework. Sure, Spring has long had fantastic features for developing amazing applications. But it was up to you to add all of the library dependencies to the project's build specification. And it was your job to write configuration to tell Spring what to do.

In this chapter, we're going to look at two ways that Spring Boot has added a level of automation to Spring development: starter dependencies and automatic configuration. You'll see how these essential Spring Boot features free you from the tedium and distraction of enabling Spring in your projects and let you focus on actually developing your applications. Along the way, you'll write a small but complete Spring application that puts Spring Boot to work for you.

2.1 *Putting Spring Boot to work*

The fact that you're reading this tells me that you are a reader. Maybe you're quite the bookworm, reading everything you can. Or maybe you only read on an as-needed basis, perhaps picking up this book only because you need to know how to develop applications with Spring.

Whatever the case may be, you're a reader. And readers tend to maintain a reading list of books that they want (or need) to read. Even if it's not a physical list, you probably have a mental list of things you'd like to read.¹

Throughout this book, we're going to build a simple reading-list application. With it, users can enter information about books they want to read, view the list, and remove books once they've been read. We'll use Spring Boot to help us develop it quickly and with as little ceremony as possible.

To start, we'll need to initialize the project. In chapter 1, we looked at a handful of ways to use the Spring Initializr to kickstart Spring Boot development. Any of those choices will work fine here, so pick the one that suits you best and get ready to put Spring Boot to work.

From a technical standpoint, we're going to use Spring MVC to handle web requests, Thymeleaf to define web views, and Spring Data JPA to persist the reading selections to a database. For now, that database will be an embedded H2 database. Although Groovy is an option, we'll write the application code in Java for now. And we'll use Gradle as our build tool of choice.

If you're using the Initializr, either via its web application or through Spring Tool Suite or IntelliJ IDEA, you'll want to be sure to select the check boxes for Web, Thymeleaf, and JPA. And also remember to check the H2 check box so that you'll have an embedded database to use while developing the application.

As for the project metadata, you're welcome to choose whatever you like. For the purposes of the reading list example, however, I created the project with the information shown in figure 2.1.

¹ If you're not a reader, feel free to apply this to movies to watch, restaurants to try, or whatever suits you.

The screenshot shows the Spring Initializr web interface in a browser window. The header says "SPRING INITIALIZR bootstrap your application now". Below the header, there's a form to generate a project. The form is divided into two main sections: "Project Metadata" and "Dependencies".

Project Metadata

- Artifact coordinates: Group (com.manning), Artifact (readinglist), Name (Reading List), Description (Reading List Demo), Package Name (readinglist), Packaging (Jar), Java Version (1.8), Language (Java).

Dependencies

- Add Spring Boot Starters and dependencies to your application.
- Search for dependencies: Web, Security, JPA, Actuator, Devtools...
- Selected Starters: Web, Thymeleaf, JPA, H2.

Too many options? [Switch back to the simple version.](#)

Generate Project

Figure 2.1 Initializing the reading list app via Initializr's web interface

If you're using Spring Tool Suite or IntelliJ IDEA to create the project, adapt the details in figure 2.1 for your IDE of choice.

On the other hand, if you're using the Spring Boot CLI to initialize the application, you can enter the following at the command line:

```
$ spring init -dweb,data-jpa,h2,thymeleaf --build gradle readinglist
```

Remember that the CLI's `init` command doesn't let you specify the project's root package or the project name. The package name will default to "demo" and the project name

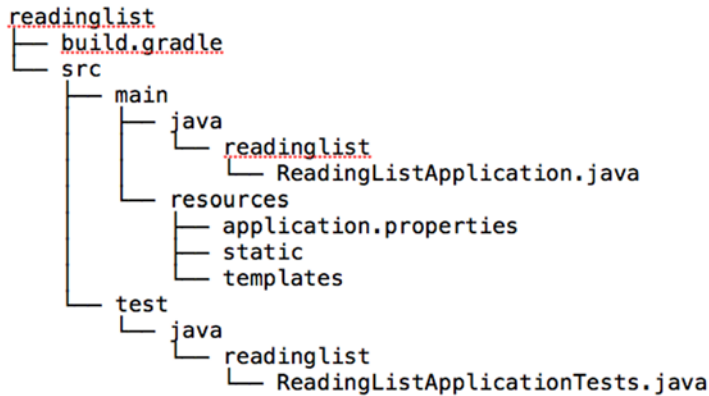


Figure 2.2 The structure of the initialized reading-list project

will default to “Demo”. After the project has been created, you’ll probably want to open it up and rename the “demo” package to “readinglist” and rename “DemoApplication.java” to “ReadingListApplication.java”.

Once the project has been created, you should have a project structure similar to that shown in figure 2.2.

This is essentially the same project structure as what the Initializr gave you in chapter 1. But now that you’re going to actually develop an application, let’s slow down and take a closer look at what’s contained in the initial project.

2.1.1 Examining a newly initialized Spring Boot project

The first thing to notice in figure 2.2 is that the project structure follows the layout of a typical Maven or Gradle project. That is, the main application code is placed in the `src/main/java` branch of the directory tree, resources are placed in the `src/main/resources` branch, and test code is placed in the `src/test/java` branch. At this point we don’t have any test resources, but if we did we’d put them in `src/test/resources`.

Digging deeper, you’ll see a handful of files sprinkled about the project:

- `build.gradle`—The Gradle build specification
- `ReadingListApplication.java`—The application’s bootstrap class and primary Spring configuration class
- `application.properties`—A place to configure application and Spring Boot properties
- `ReadingListApplicationTests.java`—A basic integration test class

There’s a lot of Spring Boot goodness to uncover in the build specification, so I’ll save inspection of it until last. Instead, we’ll start with `ReadingListApplication.java`.

BOOTSTRAPPING SPRING

The `ReadingListApplication` class serves two purposes in a Spring Boot application: configuration and bootstrapping. First, it’s the central Spring configuration class. Even though Spring Boot auto-configuration eliminates the need for a lot of

Spring configuration, you'll need at least a small amount of Spring configuration to enable auto-configuration. As you can see in listing 2.1, there's only one line of configuration code.

Listing 2.1 ReadingListApplication.java is both a bootstrap class and a configuration class

```
package readinglist;

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

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

Enable component-scanning and auto-configuration

Bootstrap the application

The `@SpringBootApplication` enables Spring component-scanning and Spring Boot auto-configuration. In fact, `@SpringBootApplication` combines three other useful annotations:

- *Spring's* `@Configuration`—Designates a class as a configuration class using Spring's Java-based configuration. Although we won't be writing a lot of configuration in this book, we'll favor Java-based configuration over XML configuration when we do.
- *Spring's* `@ComponentScan`—Enables component-scanning so that the web controller classes and other components you write will be automatically discovered and registered as beans in the Spring application context. A little later in this chapter, we'll write a simple Spring MVC controller that will be annotated with `@Controller` so that component-scanning can find it.
- *Spring Boot's* `@EnableAutoConfiguration`—This humble little annotation might as well be named `@Abracadabra` because it's the one line of configuration that enables the magic of Spring Boot auto-configuration. This one line keeps you from having to write the pages of configuration that would be required otherwise.

In older versions of Spring Boot, you'd have annotated the `ReadingListApplication` class with all three of these annotations. But since Spring Boot 1.2.0, `@SpringBootApplication` is all you need.

As I said, `ReadingListApplication` is also a bootstrap class. There are several ways to run Spring Boot applications, including traditional WAR file deployment. But for now the `main()` method here will enable you to run your application as an executable JAR file from the command line. It passes a reference to the `ReadingListApplication` class to `SpringApplication.run()`, along with the command-line arguments, to kick off the application.

In fact, even though you haven't written any application code, you can still build the application at this point and try it out. The easiest way to build and run the application is to use the `bootRun` task with Gradle:

```
$ gradle bootRun
```

The `bootRun` task comes from Spring Boot's Gradle plugin, which we'll discuss more in section 2.12. Alternatively, you can build the project with Gradle and run it with `java` at the command line:

```
$ gradle build
...
$ java -jar build/libs/readinglist-0.0.1-SNAPSHOT.jar
```

The application should start up fine and enable a Tomcat server listening on port 8080. You can point your browser at `http://localhost:8080` if you want, but because you haven't written a controller class yet, you'll be met with an HTTP 404 (Not Found) error and an error page. Before this chapter is finished, though, that URL will serve your `reading-list` application.

You'll almost never need to change `ReadingListApplication.java`. If your application requires any additional Spring configuration beyond what Spring Boot auto-configuration provides, it's usually best to write it into separate `@Configuration`-configured classes. (They'll be picked up and used by component-scanning.) In exceptionally simple cases, though, you could add custom configuration to `ReadingListApplication.java`.

TESTING SPRING BOOT APPLICATIONS

The Initializr also gave you a skeleton test class to help you get started with writing tests for your application. But `ReadingListApplicationTests` (listing 2.2) is more than just a placeholder for tests—it also serves as an example of how to write tests for Spring Boot applications.

Listing 2.2 `@SpringApplicationConfiguration` loads a Spring application context

```
package readinglist;

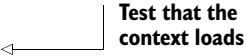
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.SpringApplicationConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import org.springframework.test.context.web.WebAppConfiguration;

import readinglist.ReadingListApplication;

@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(
    classes = ReadingListApplication.class)
@WebAppConfiguration
```

Load context via
Spring Boot

```
public class ReadingListApplicationTests {  
  
    @Test  
    public void contextLoads() {  
    }  
  
}
```



Test that the context loads

In a typical Spring integration test, you'd annotate the test class with `@ContextConfiguration` to specify how the test should load the Spring application context. But in order to take full advantage of Spring Boot magic, the `@SpringApplicationConfiguration` annotation should be used instead. As you can see from listing 2.2, `ReadingListApplicationTests` is annotated with `@SpringApplicationConfiguration` to load the Spring application context from the `ReadingListApplication` configuration class.

`ReadingListApplicationTests` also includes one simple test method, `contextLoads()`. It's so simple, in fact, that it's an empty method. But it's sufficient for the purpose of verifying that the application context loads without any problems. If the configuration defined in `ReadingListApplication` is good, the test will pass. If there are any problems, the test will fail.

Of course, you'll add some of your own tests as we flesh out the application. But the `contextLoads()` method is a fine start and verifies every bit of functionality provided by the application at this point. We'll look more at how to test Spring Boot applications in chapter 4.

CONFIGURING APPLICATION PROPERTIES

The `application.properties` file given to you by the Initializr is initially empty. In fact, this file is completely optional, so you could remove it completely without impacting the application. But there's also no harm in leaving it in place.

We'll definitely find opportunity to add entries to `application.properties` later. For now, however, if you want to poke around with `application.properties`, try adding the following line:

```
server.port=8000
```

With this line, you're configuring the embedded Tomcat server to listen on port 8000 instead of the default port 8080. You can confirm this by running the application again.

This demonstrates that the `application.properties` file comes in handy for fine-grained configuration of the stuff that Spring Boot automatically configures. But you can also use it to specify properties used by application code. We'll look at several examples of both uses of `application.properties` in chapter 3.

The main thing to notice is that at no point do you explicitly ask Spring Boot to load `application.properties` for you. By virtue of the fact that `application.properties` exists, it will be loaded and its properties made available for configuring both Spring and application code.

We're almost finished reviewing the contents of the initialized project. But we have one last artifact to look at. Let's see how a Spring Boot application is built.

2.1.2 *Dissecting a Spring Boot project build*

For the most part, a Spring Boot application isn't much different from any Spring application, which isn't much different from any Java application. Therefore, building a Spring Boot application is much like building any Java application. You have your choice of Gradle or Maven as the build tool, and you express build specifics much the same as you would in an application that doesn't employ Spring Boot. But there are a few small details about working with Spring Boot that benefit from a little extra help in the build.

Spring Boot provides build plugins for both Gradle and Maven to assist in building Spring Boot projects. Listing 2.3 shows the build.gradle file created by Initializr, which applies the Spring Boot Gradle plugin.

Listing 2.3 Using the Spring Boot Gradle plugin

```
buildscript {
    ext {
        springBootVersion = `1.3.0.RELEASE`
    }
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:
            ↳ ${springBootVersion}")
    }
}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'idea'
apply plugin: 'spring-boot'

jar {
    baseName = 'readinglist'
    version = '0.0.1-SNAPSHOT'
}
sourceCompatibility = 1.7
targetCompatibility = 1.7

repositories {
    mavenCentral()
}

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

Depend on Spring Boot plugin

Apply Spring Boot plugin

Starter dependencies

```

compile("org.springframework.boot:spring-boot-starter-thymeleaf")
runtime("com.h2database:h2")
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-1.7'
    }
}

task wrapper(type: Wrapper) {
    gradleVersion = '1.12'
}

```

On the other hand, had you chosen to build your project with Maven, the Initializr would have given you a pom.xml file that employs Spring Boot's Maven plugin, as shown in listing 2.4.

Listing 2.4 Using the Spring Boot Maven plugin and parent starter

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.manning</groupId>
  <artifactId>readinglist</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>ReadingList</name>
  <description>Reading List Demo</description>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>{springBootVersion}</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>

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

```

**Inherit versions
from starter parent**

**Starter
dependencies**

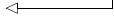

```

    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
  </dependency>
  <dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>

<properties>
  <project.build.sourceEncoding>
    UTF-8
  </project.build.sourceEncoding>
  <start-class>readinglist.Application</start-class>
  <java.version>1.7</java.version>
</properties>

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
</project>

```

**Apply Spring
Boot plugin**


Whether you choose Gradle or Maven, Spring Boot’s build plugins contribute to the build in two ways. First, you’ve already seen how you can use the `bootRun` task to run the application with Gradle. Similarly, the Spring Boot Maven plugin provides a `spring-boot:run` goal that achieves the same thing if you’re using a Maven build.

The main feature of the build plugins is that they’re able to package the project as an executable uber-JAR. This includes packing all of the application’s dependencies within the JAR and adding a manifest to the JAR with entries that make it possible to run the application with `java -jar`.

In addition to the build plugins, notice that the Maven build in listing 2.4 has “spring-boot-starter-parent” as a parent. By rooting the project in the parent starter, the build can take advantage of Maven dependency management to inherit dependency versions for several commonly used libraries so that you don’t have to explicitly specify the versions when declaring dependencies. Notice that none of the `<dependency>` entries in this `pom.xml` file specify any versions.

Unfortunately, Gradle doesn't provide the same kind of dependency management as Maven. That's why the Spring Boot Gradle plugin offers a third feature; it simulates dependency management for several common Spring and Spring-related dependencies. Consequently, the `build.gradle` file in listing 2.3 doesn't specify any versions for any of its dependencies.

Speaking of those dependencies, there are only five dependencies expressed in either build specification. And, with the exception of the H2 dependency you added manually, they all have artifact IDs that are curiously prefixed with "spring-boot-starter-". These are Spring Boot starter dependencies, and they offer a bit of build-time magic for Spring Boot applications. Let's see what benefit they provide.

2.2 Using starter dependencies

To understand the benefit of Spring Boot starter dependencies, let's pretend for a moment that they don't exist. What kind of dependencies would you add to your build without Spring Boot? Which Spring dependencies do you need to support Spring MVC? Do you remember the group and artifact IDs for Thymeleaf? Which version of Spring Data JPA should you use? Are all of these compatible?

Uh-oh. Without Spring Boot starter dependencies, you've got some homework to do. All you want to do is develop a Spring web application with Thymeleaf views that persists its data via JPA. But before you can even write your first line of code, you have to go figure out what needs to be put into the build specification to support your plan.

After much consideration (and probably a lot of copy and paste from some other application's build that has similar dependencies) you arrive at the following dependencies block in your Gradle build specification:

```
compile("org.springframework:spring-web:4.1.6.RELEASE")
compile("org.thymeleaf:thymeleaf-spring4:2.1.4.RELEASE")
compile("org.springframework.data:spring-data-jpa:1.8.0.RELEASE")
compile("org.hibernate:hibernate-entitymanager:jar:4.3.8.Final")
compile("com.h2database:h2:1.4.187")
```

This dependency list is fine and might even work. But how do you know? What kind of assurance do you have that the versions you chose for those dependencies are even compatible with each other? They might be, but you won't know until you build the application and run it. And how do you know that the list of dependencies is complete? With not a single line of code having been written, you're still a long way from kicking the tires on your build.

Let's take a step back and recall what it is we want to do. We're looking to build an application with these traits:

- It's a web application
- It uses Thymeleaf
- It persists data to a relational database via Spring Data JPA

Wouldn't it be simpler if we could just specify those facts in the build and let the build sort out what we need? That's exactly what Spring Boot starter dependencies do.

2.2.1 *Specifying facet-based dependencies*

Spring Boot addresses project dependency complexity by providing several dozen "starter" dependencies. A starter dependency is essentially a Maven POM that defines transitive dependencies on other libraries that together provide support for some functionality. Many of these starter dependencies are named to indicate the facet or kind of functionality they provide.

For example, the reading-list application is going to be a web application. Rather than add several individually chosen library dependencies to the project build, it's much easier to simply declare that this is a web application. You can do that by adding Spring Boot's web starter to the build.

We also want to use Thymeleaf for web views and persist data with JPA. Therefore, we need the Thymeleaf and Spring Data JPA starter dependencies in the build.

For testing purposes, we also want libraries that will enable us to run integration tests in the context of Spring Boot. Therefore, we also want a test-time dependency on Spring Boot's test starter.

Taken altogether, we have the following five dependencies that the Initializr provided in the Gradle build:

```
dependencies {  
    compile "org.springframework.boot:spring-boot-starter-web"  
    compile "org.springframework.boot:spring-boot-starter-thymeleaf"  
    compile "org.springframework.boot:spring-boot-starter-data-jpa"  
    compile "com.h2database:h2"  
    testCompile("org.springframework.boot:spring-boot-starter-test")  
}
```

As you saw earlier, the easiest way to get these dependencies into your application's build is to select the Web, Thymeleaf, and JPA check boxes in the Initializr. But if you didn't do that when initializing the project, you can certainly go back and add them later by editing the generated `build.gradle` or `pom.xml`.

Via transitive dependencies, adding these four dependencies is the equivalent of adding several dozen individual libraries to the build. Some of those transitive dependencies include such things as Spring MVC, Spring Data JPA, Thymeleaf, as well as any transitive dependencies that those dependencies declare.

The most important thing to notice about the four starter dependencies is that they were only as specific as they needed to be. We didn't say that we wanted Spring MVC; we simply said we wanted to build a web application. We didn't specify JUnit or any other testing tools; we just said we wanted to test our code. The Thymeleaf and Spring Data JPA starters are a bit more specific, but only because there's no less-specific way to declare that you want Thymeleaf and Spring Data JPA.

The four starters in this build are only a few of the many starter dependencies that Spring Boot offers. Appendix B lists all of the starters with some detail on what each one transitively brings to a project build.

In no case did we need to specify the version. The versions of the starter dependencies themselves are determined by the version of Spring Boot you're using. The starter dependencies themselves determine the versions of the various transitive dependencies that they pull in.

Not knowing what versions of the various libraries are used may be a little unsettling to you. Be encouraged to know that Spring Boot has been tested to ensure that all of the dependencies pulled in are compatible with each other. It's actually very liberating to just specify a starter dependency and not have to worry about which libraries and which versions of those libraries you need to maintain.

But if you really must know what it is that you're getting, you can always get that from the build tool. In the case of Gradle, the `dependencies` task will give you a dependency tree that includes every library your project is using and their versions:

```
$ gradle dependencies
```

You can get a similar dependency tree from a Maven build with the `tree` goal of the `dependency` plugin:

```
$ mvn dependency:tree
```

For the most part, you should never concern yourself with the specifics of what each Spring Boot starter dependency provides. Generally, it's enough to know that the web starter enables you to build a web application, the Thymeleaf starter enables you to use Thymeleaf templates, and the Spring Data JPA starter enables data persistence to a database using Spring Data JPA.

But what if, in spite of the testing performed by the Spring Boot team, there's a problem with a starter dependency's choice of libraries? How can you override the starter?

2.2.2 Overriding starter transitive dependencies

Ultimately, starter dependencies are just dependencies like any other dependency in your build. That means you can use the facilities of the build tool to selectively override transitive dependency versions, exclude transitive dependencies, and certainly specify dependencies for libraries not covered by Spring Boot starters.

For example, consider Spring Boot's web starter. Among other things, the web starter transitively depends on the Jackson JSON library. This library is handy if you're building a REST service that consumes or produces JSON resource representations. But if you're using Spring Boot to build a more traditional human-facing web application, you may not need Jackson. Even though it shouldn't hurt anything to include it, you can trim the fat off of your build by excluding Jackson as a transitive dependency.

If you're using Gradle, you can exclude transitive dependencies like this:

```
compile("org.springframework.boot:spring-boot-starter-web") {  
    exclude group: 'com.fasterxml.jackson.core'  
}
```

In Maven, you can exclude transitive dependencies with the `<exclusions>` element. The following `<dependency>` for the Spring Boot web starter has `<exclusions>` to keep Jackson out of the build:

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-web</artifactId>  
  <exclusions>  
    <exclusion>  
      <groupId>com.fasterxml.jackson.core</groupId>  
    </exclusion>  
  </exclusions>  
</dependency>
```

On the other hand, maybe having Jackson in the build is fine, but you want to build against a different version of Jackson than what the web starter references. Suppose that the web starter references Jackson version 2.3.4, but you'd rather user version 2.4.3.² Using Maven, you can express the desired dependency directly in your project's pom.xml file like this:

```
<dependency>  
  <groupId>com.fasterxml.jackson.core</groupId>  
  <artifactId>jackson-databind</artifactId>  
  <version>2.4.3</version>  
</dependency>
```

Maven always favors the closest dependency, meaning that because you've expressed this dependency in your project's build, it will be favored over the one that's transitively referred to by another dependency.

Similarly, if you're building with Gradle, you can specify the newer version of Jackson in your build.gradle file like this:

```
compile("com.fasterxml.jackson.core:jackson-databind:2.4.3")
```

This dependency works in Gradle because it's newer than the version transitively referred to by Spring Boot's web starter. But suppose that instead of using a newer version of Jackson, you'd like to use an older version. Unlike Maven, Gradle favors the newest version of a dependency. Therefore, if you want to use an older version of

² The versions mentioned here are for illustration purposes only. The actual version of Jackson referenced by Spring Boot's web starter will be determined by which version of Spring Boot you are using.

Jackson, you'll have to express the older version as a dependency in your build and exclude it from being transitively resolved by the web starter dependency:

```
compile("org.springframework.boot:spring-boot-starter-web") {  
    exclude group: 'com.fasterxml.jackson.core'  
}  
compile("com.fasterxml.jackson.core:jackson-databind:2.3.1")
```

In any case, take caution when overriding the dependencies that are pulled in transitively by Spring Boot starter dependencies. Although different versions may work fine, there's a great amount of comfort that can be taken knowing that the versions chosen by the starters have been tested to play well together. You should only override these transitive dependencies under special circumstances (such as a bug fix in a newer version).

Now that we have an empty project structure and build specification ready, it's time to start developing the application itself. As we do, we'll let Spring Boot handle the configuration details while we focus on writing the code that provides the reading-list functionality.

2.3 Using automatic configuration

In a nutshell, Spring Boot auto-configuration is a runtime (more accurately, application startup-time) process that considers several factors to decide what Spring configuration should and should not be applied. To illustrate, here are a few examples of the kinds of things that Spring Boot auto-configuration might consider:

- Is Spring's `JdbcTemplate` available on the classpath? If so and if there is a `DataSource` bean, then auto-configure a `JdbcTemplate` bean.
- Is Thymeleaf on the classpath? If so, then configure a Thymeleaf template resolver, view resolver, and template engine.
- Is Spring Security on the classpath? If so, then configure a very basic web security setup.

There are nearly 200 such decisions that Spring Boot makes with regard to auto-configuration every time an application starts up, covering such areas as security, integration, persistence, and web development. All of this auto-configuration serves to keep you from having to explicitly write configuration unless absolutely necessary.

The funny thing about auto-configuration is that it's difficult to show in the pages of this book. If there's no configuration to write, then what is there to point to and discuss?

2.3.1 Focusing on application functionality

One way to gain an appreciation of Spring Boot auto-configuration would be for me to spend the next several pages showing you the configuration that's required in the absence of Spring Boot. But there are already several great books on Spring that show