Spring Boot

IN ACTION

Craig Walls

Foreword by Andrew Glover

SAMPLE CHAPTER

MANNING

Spring Boot in Action

by Craig Walls

Chapter 2

brief content

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.

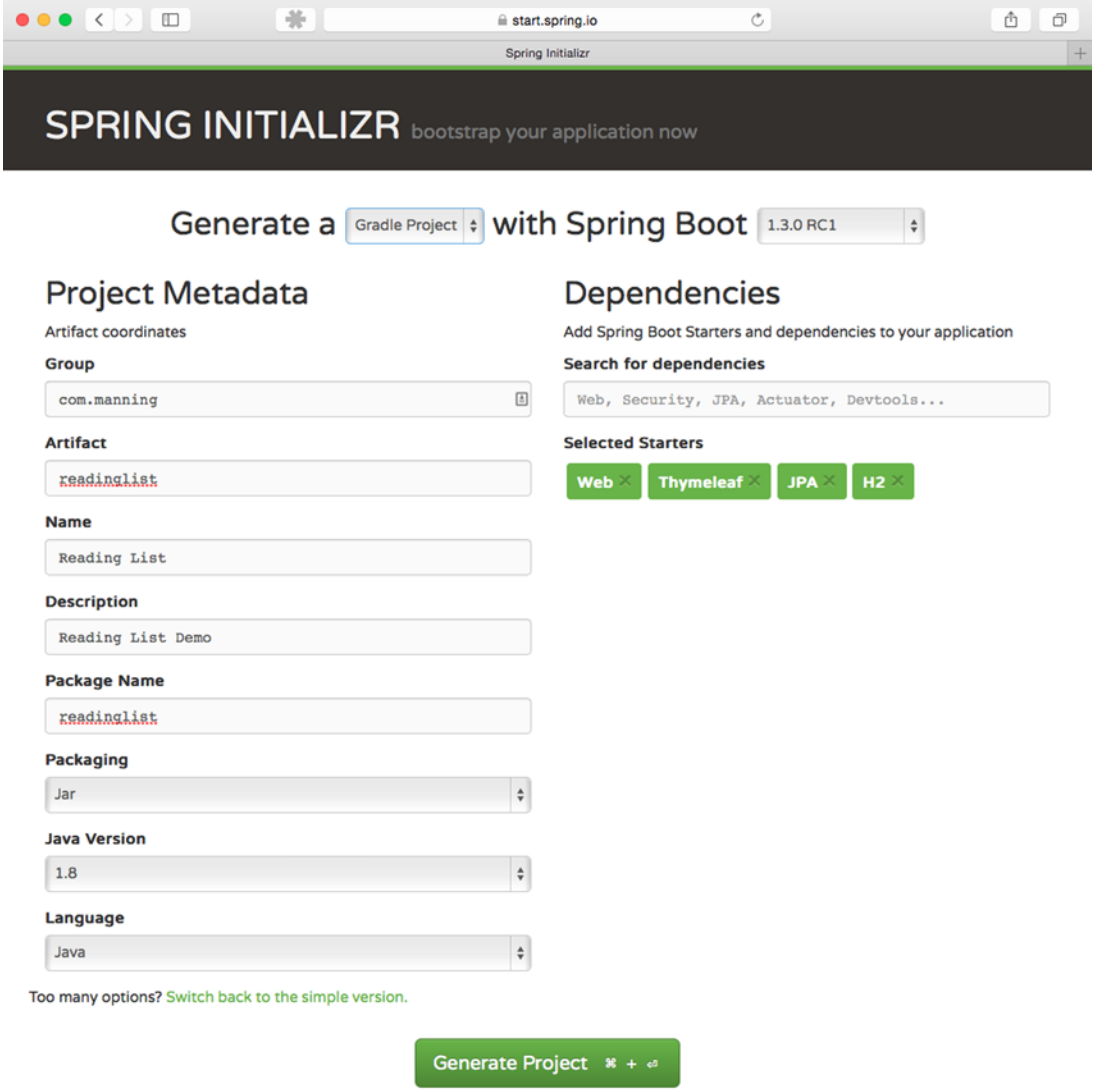
To start, we'll need to initialize the project. In chapter 1, we looked at a handful of ways to use the Spring Initializr to kickstart Spring Boot development. Any of those choices will work fine here, so pick the one that suits you best and get ready to put Spring Boot to work.

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.



Generate a Gradle Project with Spring Boot 1.3.0 RC1

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

Too many options? Switch back to the simple version.

Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies: Web, Security, JPA, Actuator, Devtools...

Selected Starters: Web ✕ Thymeleaf ✕ JPA ✕ H2 ✕

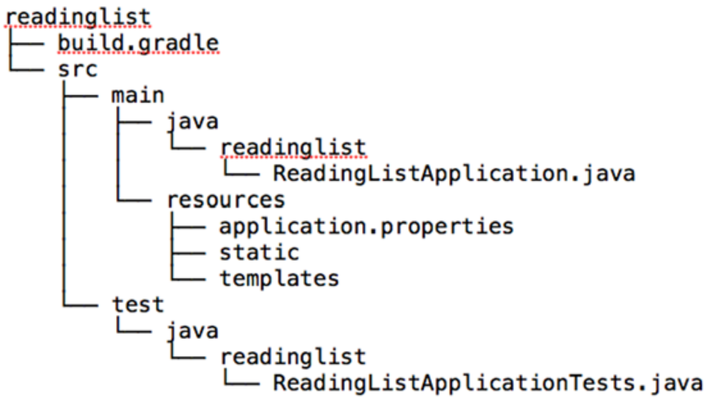
Figure 2.1 Initializing the reading list app via Initializr's web interface

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



```
readinglist
├── build.gradle
└── src
    ├── main
    │   ├── java
    │   │   └── readinglist
    │   │       └── ReadingListApplication.java
    │   └── resources
    │       ├── application.properties
    │       ├── static
    │       └── templates
    └── test
        └── java
            └── readinglist
                └── ReadingListApplicationTests.java
```

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

  <dependencies>                                                   Starter
    <dependency>                                                   dependencies
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <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 you that, and showing it again wouldn't help us get the reading-list application written any quicker.

Instead of wasting time talking about Spring configuration, knowing that Spring Boot is going to take care of that for us, let's see how taking advantage of Spring Boot auto-configuration keeps us focused on writing application code. I can think of no better way to do that than to start writing the application code for the reading-list application.

DEFINING THE DOMAIN

The central domain concept in our application is a book that's on a reader's reading list. Therefore, we'll need to define an entity class that represents a book. Listing 2.5 shows how the `Book` type is defined.

Listing 2.5 The `Book` class represents a book in the reading list

```
package readinglist;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Book {

  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private Long id;
  private String reader;
  private String isbn;
  private String title;
  private String author;
  private String description;

  public Long getId() {
    return id;
  }

  public void setId(Long id) {
    this.id = id;
  }

  public String getReader() {
    return reader;
  }

  public void setReader(String reader) {
    this.reader = reader;
  }

  public String getIsbn() {
    return isbn;
  }

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

  public String getTitle() {
    return title;
  }

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

  public String getAuthor() {
    return author;
  }

  public void setAuthor(String author) {
    this.author = author;
  }

  public String getDescription() {
    return description;
  }

  public void setDescription(String description) {
    this.description = description;
  }

}
```

As you can see, the Book class is a simple Java object with a handful of properties describing a book and the necessary accessor methods. It's annotated with @Entity designating it as a JPA entity. The id property is annotated with @Id and @GeneratedValue to indicate that this field is the entity's identity and that its value will be automatically provided.

DEFINING THE REPOSITORY INTERFACE

Next up, we need to define the repository through which the ReadingList objects will be persisted to the database. Because we're using Spring Data JPA, that task is a simple matter of creating an interface that extends Spring Data JPA's JpaRepository interface:

```
package readinglist;

import java.util.List;
import org.springframework.data.jpa.repository.JpaRepository;

public interface ReadingListRepository extends JpaRepository<Book, Long> {

  List<Book> findByReader(String reader);

}
```

By extending JpaRepository, ReadingListRepository inherits 18 methods for performing common persistence operations. The JpaRepository interface is parameterized with two parameters: the domain type that the repository will work with, and the type of its ID property. In addition, I've added a findByReader() method through which a reading list can be looked up given a reader's username.

If you're wondering about who will implement ReadingListRepository and the 18 methods it inherits, don't worry too much about it. Spring Data provides a special magic of its own, making it possible to define a repository with just an interface. The interface will be implemented automatically at runtime when the application is started.

CREATING THE WEB INTERFACE

Now that we have the application's domain defined and a repository for persisting objects from that domain to the database, all that's left is to create the web front-end. A Spring MVC controller like the one in listing 2.6 will handle HTTP requests for the application.

Listing 2.6 A Spring MVC controller that fronts the reading list application

```
package readinglist;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

import java.util.List;

@Controller
@RequestMapping("/")
public class ReadingListController {

  private ReadingListRepository readingListRepository;

  @Autowired
  public ReadingListController(
             ReadingListRepository readingListRepository) {
    this.readingListRepository = readingListRepository;
  }

  @RequestMapping(value="/{reader}", method=RequestMethod.GET)
  public String readersBooks(
      @PathVariable("reader") String reader,
      Model model) {

    List<Book> readingList =
        readingListRepository.findByReader(reader);
    if (readingList != null) {
      model.addAttribute("books", readingList);
    }
    return "readingList";
  }
  @RequestMapping(value="/{reader}", method=RequestMethod.POST)
  public String addToReadingList(
            @PathVariable("reader") String reader, Book book) {
    book.setReader(reader);
    readingListRepository.save(book);
    return "redirect:/{reader}";
  }

}
```

ReadingListController is annotated with @Controller in order to be picked up by component-scanning and automatically be registered as a bean in the Spring application context. It's also annotated with @RequestMapping to map all of its handler methods to a base URL path of "/".

The controller has two methods:

- readersBooks()—Handles HTTP GET requests for /{reader} by retrieving a Book list from the repository (which was injected into the controller's constructor) for the reader specified in the path. It puts the list of Book into the model under the key "books" and returns "readingList" as the logical name of the view to render the model.
- addToReadingList()—Handles HTTP POST requests for /{reader}, binding the data in the body of the request to a Book object. This method sets the Book object's reader property to the reader's name, and then saves the modified Book via the repository's save() method. Finally, it returns by specifying a redirect to /{reader} (which will be handled by the other controller method).

The readersBooks() method concludes by returning "readingList" as the logical view name. Therefore, we must also create that view. I decided at the outset of this project that we'd be using Thymeleaf to define the application views, so the next step is to create a file named readingList.html in src/main/resources/templates with the following content.

Listing 2.7 The Thymeleaf template that presents a reading list

```
<html>
  <head>
    <title>Reading List</title>
    <link rel="stylesheet" th:href="@{/style.css}"></link>
  </head>

  <body>
    <h2>Your Reading List</h2>
    <div th:unless="${#lists.isEmpty(books)}">
      <dl th:each="book : ${books}">
        <dt class="bookHeadline">
          <span th:text="${book.title}">Title</span> by
          <span th:text="${book.author}">Author</span>
          (ISBN: <span th:text="${book.isbn}">ISBN</span>)
        </dt>
        <dd class="bookDescription">
          <span th:if="${book.description}"
                th:text="${book.description}">Description</span>
          <span th:if="${book.description eq null}">
                No description available</span>
        </dd>
      </dl>
    </div>
    <div th:if="${#lists.isEmpty(books)}">
      <p>You have no books in your book list</p>
    </div>

    <hr/>

    <h3>Add a book</h3>
    <form method="POST">
      <label for="title">Title:</label>
        <input type="text" name="title" size="50"></input><br/>
      <label for="author">Author:</label>
        <input type="text" name="author" size="50"></input><br/>
      <label for="isbn">ISBN:</label>
        <input type="text" name="isbn" size="15"></input><br/>
      <label for="description">Description:</label><br/>
        <textarea name="description" cols="80" rows="5">
        </textarea><br/>
      <input type="submit"></input>
    </form>

  </body>
</html>
```

This template defines an HTML page that is conceptually divided into two parts. At the top of the page is a list of books that are in the reader's reading list. At the bottom is a form the reader can use to add a new book to the reading list.

For aesthetic purposes, the Thymeleaf template references a stylesheet named style.css. That file should be created in src/main/resources/static and look like this:

```
body {
    background-color: #cccccc;
    font-family: arial,helvetica,sans-serif;
}

.bookHeadline {
    font-size: 12pt;
    font-weight: bold;
}

.bookDescription {
    font-size: 10pt;
}

label {
    font-weight: bold;
}
```

This stylesheet is simple and doesn't go overboard to make the application look nice. But it serves our purposes and, as you'll soon see, serves to demonstrate a piece of Spring Boot's auto-configuration.

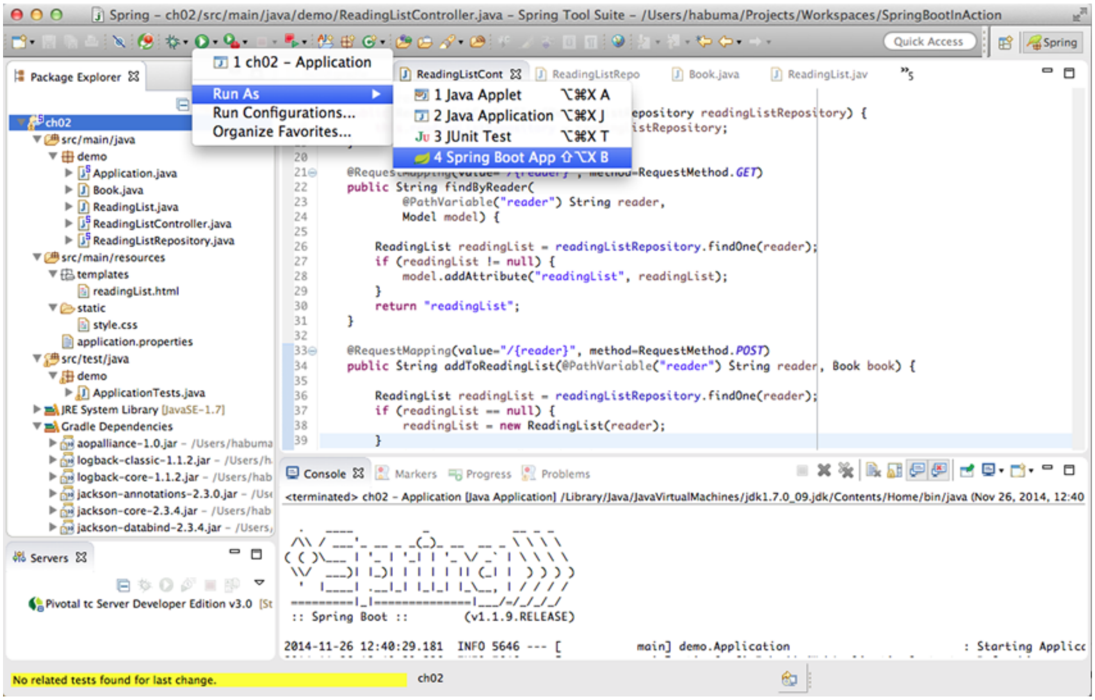
Believe it or not, that's a complete application. Every single line has been presented to you in this chapter. Take a moment, flip back through the previous pages, and see if you can find any configuration. In fact, aside from the three lines of configuration in listing 2.1 (which essentially turn on auto-configuration), you didn't have to write any Spring configuration.

Despite the lack of Spring configuration, this complete Spring application is ready to run. Let's fire it up and see how it looks.

2.3.2 Running the application

There are several ways to run a Spring Boot application. Earlier, in section 2.5, we discussed how to run the application via Maven and Gradle, as well as how to build and run an executable JAR. Later, in chapter 8 you'll also see how to build a WAR file that can be deployed in a traditional manner to a Java web application server such as Tomcat.

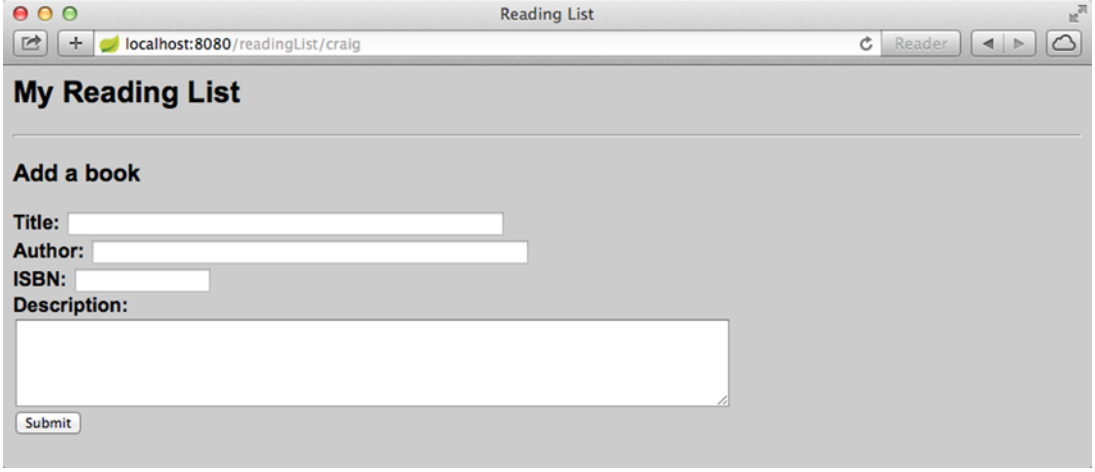
If you're developing your application with Spring Tool Suite, you also have the option of running the application within your IDE by selecting the project and choosing Run As > Spring Boot App from the Run menu, as shown in figure 2.3.


Spring – ch02/src/main/java/demo/ReadingListController.java – Spring Tool Suite – /Users/habuma/Projects/Workspaces/SpringBootInAction
Quick Access
Spring
Package Explorer
ch02
src/main/java
demo
Application.java
Book.java
ReadingList.java
ReadingListController.java
ReadingListRepository.java
src/main/resources
templates
readingList.html
static
style.css
application.properties
src/test/java
demo
ApplicationTests.java
JRE System Library [JavaSE-1.7]
Gradle Dependencies
aopalliance-1.0.jar – /Users/habuma
logback-classic-1.1.2.jar – /Users/h
logback-core-1.1.2.jar – /Users/hab
jackson-annotations-2.3.0.jar – /Us
jackson-core-2.3.4.jar – /Users/hab
jackson-databind-2.3.4.jar – /Users,
Servers
Pivotal tc Server Developer Edition v3.0 [St
1 ch02 – Application
Run As
Run Configurations...
Organize Favorites...
1 Java Applet ⌥⌘X A
2 Java Application ⌥⌘X J
3 JUnit Test ⌥⌘X T
4 Spring Boot App ⇧⌥X B
ReadingListCont
ReadingListRepo
Book.java
ReadingList.jav
Console
Markers
Progress
Problems
No related tests found for last change.
ch02

```
                                    epository readingListRepository) {
                                    istRepository;

    @Reques                       , method=RequestMethod.GET)
    public String findByReader(
            @PathVariable("reader") String reader,
            Model model) {

        ReadingList readingList = readingListRepository.findOne(reader);
        if (readingList != null) {
            model.addAttribute("readingList", readingList);
        }
        return "readingList";
    }

    @RequestMapping(value="/{reader}", method=RequestMethod.POST)
    public String addToReadingList(@PathVariable("reader") String reader, Book book) {

        ReadingList readingList = readingListRepository.findOne(reader);
        if (readingList == null) {
            readingList = new ReadingList(reader);
        }
```

```
<terminated> ch02 – Application [Java Application] /Library/Java/JavaVirtualMachines/jdk1.7.0_09.jdk/Contents/Home/bin/java (Nov 26, 2014, 12:40

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.1.9.RELEASE)

2014-11-26 12:40:29.181  INFO 5646 --- [           main] demo.Application                         : Starting Applic
```

Figure 2.3 Running a Spring Boot application from Spring Tool Suite



My Reading List

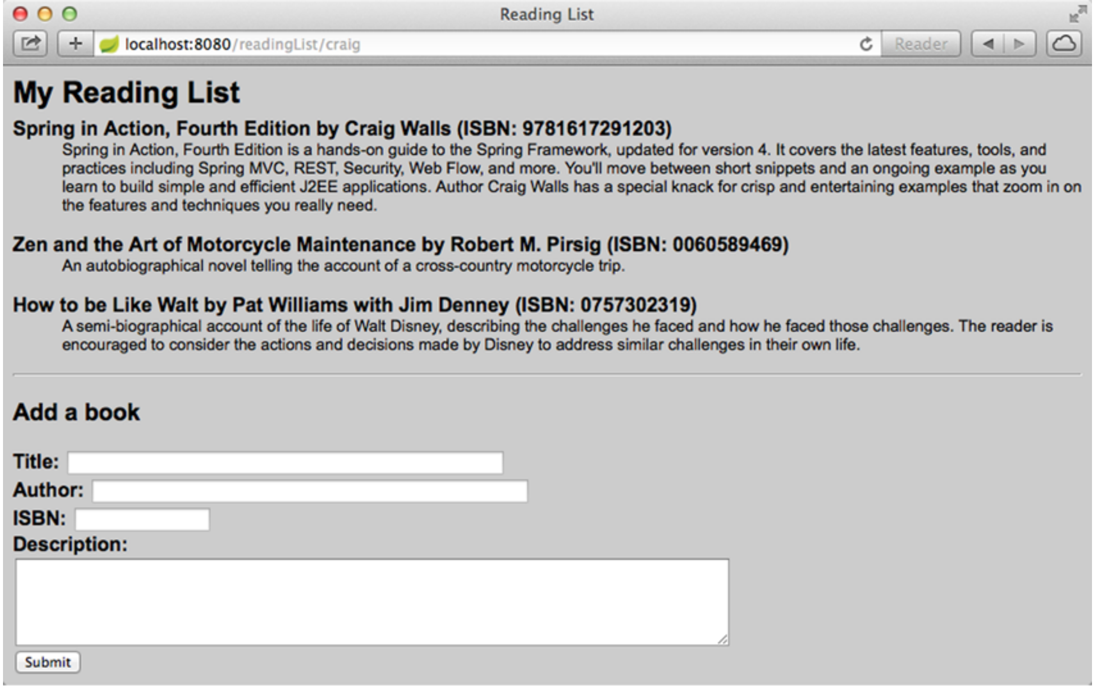
Add a book

Title:
Author:
ISBN:
Description:

Figure 2.4 An initially empty reading list

Assuming everything works, your browser should show you an empty reading list along with a form for adding a new book to the list. Figure 2.4 shows what it might look like.

Now go ahead and use the form to add a few books to your reading list. After you do, your list might look something like figure 2.5.



My Reading List

Spring in Action, Fourth Edition by Craig Walls (ISBN: 9781617291203)
Spring in Action, Fourth Edition is a hands-on guide to the Spring Framework, updated for version 4. It covers the latest features, tools, and practices including Spring MVC, REST, Security, Web Flow, and more. You'll move between short snippets and an ongoing example as you learn to build simple and efficient J2EE applications. Author Craig Walls has a special knack for crisp and entertaining examples that zoom in on the features and techniques you really need.

Zen and the Art of Motorcycle Maintenance by Robert M. Pirsig (ISBN: 0060589469)
An autobiographical novel telling the account of a cross-country motorcycle trip.

How to be Like Walt by Pat Williams with Jim Denney (ISBN: 0757302319)
A semi-biographical account of the life of Walt Disney, describing the challenges he faced and how he faced those challenges. The reader is encouraged to consider the actions and decisions made by Disney to address similar challenges in their own life.

Add a book

Title:
Author:
ISBN:
Description:

Figure 2.5 The reading list after a few books have been added

Feel free to take a moment to play around with the application. When you're ready, move on and we'll see how Spring Boot made it possible to write an entire Spring application with no Spring configuration code.

2.3.3 What just happened?

As I said, it's hard to describe auto-configuration when there's no configuration to point at. So instead of spending time discussing what you don't have to do, this section has focused on what you do need to do—namely, write the application code.

But certainly there is some configuration somewhere, right? Configuration is a central element of the Spring Framework, and there must be something that tells Spring how to run your application.

When you add Spring Boot to your application, there's a JAR file named spring-boot-autoconfigure that contains several configuration classes. Every one of these configuration classes is available on the application's classpath and has the opportunity to contribute to the configuration of your application. There's configuration for Thymeleaf, configuration for Spring Data JPA, configuration for Spring MVC, and configuration for dozens of other things you might or might not want to take advantage of in your Spring application.

What makes all of this configuration special, however, is that it leverages Spring's support for conditional configuration, which was introduced in Spring 4.0. Conditional configuration allows for configuration to be available in an application, but to be ignored unless certain conditions are met.

It's easy enough to write your own conditions in Spring. All you have to do is implement the `Condition` interface and override its `matches()` method. For example, the following simple condition class will only pass if `JdbcTemplate` is available on the classpath:

```
package readinglist;
import org.springframework.context.annotation.Condition;
import org.springframework.context.annotation.ConditionContext;
import org.springframework.core.type.AnnotatedTypeMetadata;

public class JdbcTemplateCondition implements Condition {
  @Override
  public boolean matches(ConditionContext context,
                         AnnotatedTypeMetadata metadata) {
    try {
      context.getClassLoader().loadClass(
             "org.springframework.jdbc.core.JdbcTemplate");
      return true;
    } catch (Exception e) {
      return false;
    }
  }
}
```

You can use this custom condition class when you declare beans in Java:

```
@Conditional(JdbcTemplateCondition.class)
public MyService myService() {
    ...
}
```

In this case, the `MyService` bean will only be created if the `JdbcTemplateCondition` passes. That is to say that the `MyService` bean will only be created if `JdbcTemplate` is available on the classpath. Otherwise, the bean declaration will be ignored.

Although the condition shown here is rather simple, Spring Boot defines several more interesting conditions and applies them to the configuration classes that make up Spring Boot auto-configuration. Spring Boot applies conditional configuration by defining several special conditional annotations and using them in its configuration classes. Table 2.1 lists the conditional annotations that Spring Boot provides.

Table 2.1 Conditional annotations used in auto-configuration

Conditional annotation	Configuration applied if…?
`@ConditionalOnBean`	…the specified bean has been configured
`@ConditionalOnMissingBean`	…the specified bean has not already been configured
`@ConditionalOnClass`	…the specified class is available on the classpath
`@ConditionalOnMissingClass`	…the specified class is not available on the classpath
`@ConditionalOnExpression`	…the given Spring Expression Language (SpEL) expression evaluates to `true`
`@ConditionalOnJava`	…the version of Java matches a specific value or range of versions
`@ConditionalOnJndi`	…there is a JNDI `InitialContext` available and optionally given JNDI locations exist
`@ConditionalOnProperty`	…the specified configuration property has a specific value
`@ConditionalOnResource`	…the specified resource is available on the classpath
`@ConditionalOnWebApplication`	…the application is a web application
`@ConditionalOnNotWebApplication`	…the application is not a web application

Generally, you shouldn't ever need to look at the source code for Spring Boot's auto-configuration classes. But as an illustration of how the annotations in table 2.1 are used, consider this excerpt from `DataSourceAutoConfiguration` (provided as part of Spring Boot's auto-configuration library):

```
@Configuration
@ConditionalOnClass({ DataSource.class, EmbeddedDatabaseType.class })
@EnableConfigurationProperties(DataSourceProperties.class)
```

```
@Import({ Registrar.class, DataSourcePoolMetadataProvidersConfiguration.class
      })
public class DataSourceAutoConfiguration {

...

}
```

As you can see, DataSourceAutoConfiguration is a @Configuration-annotated class that (among other things) imports some additional configuration from other configuration classes and defines a few beans of its own. What's most important to notice here is that DataSourceAutoConfiguration is annotated with @ConditionalOnClass to require that both DataSource and EmbeddedDatabaseType be available on the classpath. If they aren't available, then the condition fails and any configuration provided by DataSourceAutoConfiguration will be ignored.

Within DataSourceAutoConfiguration there's a nested JdbcTemplateConfiguration class that provides auto-configuration of a JdbcTemplate bean:

```
@Configuration
@Conditional(DataSourceAutoConfiguration.DataSourceAvailableCondition.class)
protected static class JdbcTemplateConfiguration {

  @Autowired(required = false)
  private DataSource dataSource;

  @Bean
  @ConditionalOnMissingBean(JdbcOperations.class)
  public JdbcTemplate jdbcTemplate() {
    return new JdbcTemplate(this.dataSource);
  }

...

}
```

JdbcTemplateConfiguration is an annotation with the low-level @Conditional to require that the DataSourceAvailableCondition pass—essentially requiring that a DataSource bean be available or that one will be created by auto-configuration. Assuming that a DataSource bean will be available, the @Bean-annotated jdbcTemplate() method configures a JdbcTemplate bean. But jdbcTemplate() is annotated with @ConditionalOnMissingBean so that the bean will be configured only if there is not already a bean of type JdbcOperations (the interface that JdbcTemplate implements).

There's a lot more to DataSourceAutoConfiguration and to the other auto-configuration classes provided by Spring Boot than is shown here. But this should give you a taste of how Spring Boot leverages conditional configuration to implement auto-configuration.

As it directly pertains to our example, the following configuration decisions are made by the conditionals in auto-configuration:

- Because H2 is on the classpath, an embedded H2 database bean will be created. This bean is of type `javax.sql.DataSource`, which the JPA implementation (Hibernate) will need to access the database.
- Because Hibernate Entity Manager is on the classpath (transitively via Spring Data JPA), auto-configuration will configure beans needed to support working with Hibernate, including Spring's `LocalContainerEntityManagerFactoryBean` and `JpaVendorAdapter`.
- Because Spring Data JPA is on the classpath, Spring Data JPA will be configured to automatically create repository implementations from repository interfaces.
- Because Thymeleaf is on the classpath, Thymeleaf will be configured as a view option for Spring MVC, including a Thymeleaf template resolver, template engine, and view resolver. The template resolver is configured to resolve templates from /templates relative to the root of the classpath.
- Because Spring MVC is on the classpath (thanks to the web starter dependency), Spring's `DispatcherServlet` will be configured and Spring MVC will be enabled.
- Because this is a Spring MVC web application, a resource handler will be registered to serve static content from /static relative to the root of the classpath. (The resource handler will also serve static content from /public, /resources, and /META-INF/resources).
- Because Tomcat is on the classpath (transitively referred to by the web starter dependency), an embedded Tomcat container will be started to listen on port 8080.

The main takeaway here, though, is that Spring Boot auto-configuration takes on the burden of configuring Spring so that you can focus on writing your application.

2.4 Summary

By taking advantage of Spring Boot starter dependencies and auto-configuration, you can more quickly and easily develop Spring applications. Starter dependencies help you focus on the type of functionality your application needs rather than on the specific libraries and versions that provide that functionality. Meanwhile, auto-configuration frees you from the boilerplate configuration that is common among Spring applications without Spring Boot.

Although auto-configuration is a convenient way to work with Spring, it also represents an opinionated approach to Spring development. What if you want or need to configure Spring differently? In the next chapter, we'll look at how you can override Spring Boot auto-configuration as needed to achieve the goals of your application. You'll also see how to apply some of the same techniques to configure your own application components.

JAVA

Spring Boot IN ACTION

Craig Walls

The Spring Framework simplifies enterprise Java development, but it does require lots of tedious configuration work. Spring Boot radically streamlines spinning up a Spring application. You get automatic configuration and a model with established conventions for build-time and runtime dependencies. You also get a handy command-line interface you can use to write scripts in Groovy. Developers who use Spring Boot often say that they can't imagine going back to hand configuring their applications.

Spring Boot in Action is a developer-focused guide to writing applications using Spring Boot. In it, you'll learn how to bypass configuration steps so you can focus on your application's behavior. Spring expert Craig Walls uses interesting and practical examples to teach you both how to use the default settings effectively and how to override and customize Spring Boot for your unique environment. Along the way, you'll pick up insights from Craig's years of Spring development experience.

What's Inside

- Develop Spring apps more efficiently
- Minimal to no configuration
- Runtime metrics with the Actuator
- Covers Spring Boot 1.3

Written for readers familiar with the Spring Framework.

Craig Walls is a software developer, author of the popular book *Spring in Action, Fourth Edition*, and a frequent speaker at conferences.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/spring-boot-in-action

Free eBook
SEE INSERT

“Easy to digest and fun to read.”

—From the Foreword by Andrew Glover, Netflix

“The evolution of Spring continues, and this guide helps maximize its potential.”

—Michael A. Angelo
ThreatConnect

“A lucid, real-world treatment of a valuable toolset. The practical examples help bring agility and simplicity to application construction.”

—Eric Kramer
Research Institute at Nationwide Children's Hospital

“Easy-to-follow, comprehensive, awesome!”

—Furkan Kamaci, Alcatel-Lucent

ISBN 13: 978-1-61729-254-5
ISBN 10: 1-61729-254-0
54499
9 781617 292545

MANNING

$44.99 / Can $51.99 [INCLUDING eBOOK]