■ Automatic configuration—Spring Boot can automatically provide configuration for application functionality common to many Spring applications.

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

@Bean

public JdbcTemplate jdbcTemplate(DataSource dataSource)

{

return new JdbcTemplate(dataSource);

}

@Bean

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

}

Any application that needs an embedded database and a JdbcTemplate will need those methods. In short, it’s boilerplate configuration.

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.

■ 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.

t 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

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.

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.

■ 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.

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. I 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.

Getting started with Spring Boot

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

$ spring –version

Spring Initializr can be used in several ways:

■ Through a web-based interface (to http://start.spring.io)

■ Via Spring Tool Suite

■ Via IntelliJ IDEA

■ Using the Spring Boot CLI

$ spring init

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

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

$ 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

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

Examining a newly initialized Spring Boot project

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); }

}

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

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.

$ gradle bootRun

Mvn spring-boot:run (maven goal)

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

If your application requires any additional Spring configuration beyond what Spring Boot auto-configuration provides, it’s usually best to write it into separate @Configurationconfigured classes. (They’ll be picked up and used by component-scanning.)

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

Listing 2.2 @SpringApplicationConfiguration loads a Spring application context

Load context via Spring Boot

Licensed to Thomas Snead <n.ordickan@gmail.com> www.it-ebooks.info

29Putting Spring Boot to work

public class ReadingListApplicationTests {

@Test public void contextLoads() { }

}

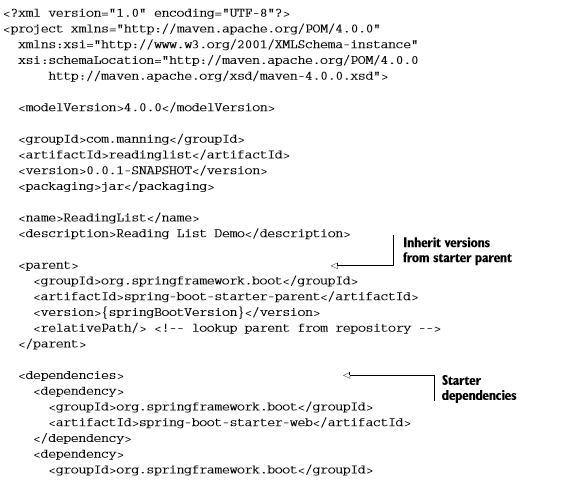
We’ll definitely find opportunity to add entries to application.properties later. For now, however, if you want to poke around with application.properties, try adding the following line:

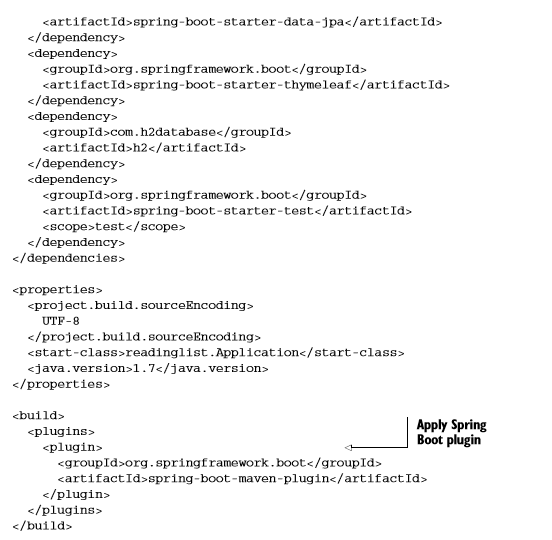
server.port=8000

With this line, you’re configuring the embedded Tomcat server to listen on port 8000 instead of the default port 8080

Maven pom file

Spring war mywar.war myapp.groovy





Overriding dependency

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.

<?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.example</groupId>

<artifactId>myproject</artifactId>

<version>0.0.1-SNAPSHOT</version>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.3.5.RELEASE</version>

</parent>

<!-- Additional lines to be added here... -->

</project>

Mvn pcakge

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter</artifactId>

<exclusions>

<exclusion>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-logging</artifactId>

</exclusion>

</exclusions>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-log4j</artifactId>

</dependency>

LESSON 1 RESTFUL SERVICE