Spring Boot basics

Get started using Spring Boot to write Spring applications that 'just run'

J Steven Perry

May 11, 2017

Learn how to use Spring Boot's starters, opinions, and executable JAR file structure to quickly write Spring-based applications that "just run."

Spring Boot is a lightweight framework that takes most of the work out of configuring Spring-based applications. In this tutorial, you'll learn how to use Spring Boot's starters, opinions, and executable JAR file structure to quickly create Spring-based applications that "just run."

After briefly introducing Spring Boot, I'll guide you through setting up and running two Spring Boot applications: a simple "Hello, World" app, and a slightly more complex Spring MVC RESTful web services application. Note that I use Spring Boot version 1.5.2 for my application examples. I recommend you use 1.5.2 or higher, but the examples should work for any release after 1.5.

Prerequisites

To get the most out of the tutorial, you should be comfortable using the Java™ Development Kit, version 8 (JDK 8). You'll also need to be familiar with Eclipse IDE, Maven, and Git.

To follow along with the tutorial, be sure this software is installed:

- JDK 8 for Windows, Mac, and Linux.
- Eclipse IDE for Windows, Mac, and Linux.
- Apache Maven for Windows, Mac, and Linux.
- Git for Windows, Mac, and Linux.

Before we dive into the tutorial, I want to talk a little about Spring Boot.

What is Spring Boot?

The goal of Spring Boot is to provide a set of tools for quickly building Spring applications that are easy to configure.

The problem: Configuring Spring is hard!

If you've ever written a Spring-based application, you know that a lot of work goes into configuring it just to get to "Hello, World." This isn't a bad thing: Spring is an elegant set of frameworks that require carefully coordinated configuration to work correctly. But that elegance comes at the cost of configuration complexity (and don't even get me started on all that XML).

The solution: Spring Boot

Enter Spring Boot. The Spring Boot website says it much more succinctly than I can:

> *“ Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run." We take an opinionated view of the Spring platform and third-party libraries so you can get started with minimum fuss. ”*

Basically, this means you can quickly get a Spring application up and running with very little configuration. What little configuration there is comes in the form of annotations, so no XML.

All of that sounds great, right? But how exactly does Spring Boot work?

First, it's opinionated

Spring Boot has *opinions*. This is just another way of saying that Spring Boot has *reasonable defaults*, so you can build an application quickly using these commonly used values.

As an example, Tomcat is a very popular web container. By default, a Spring Boot web application uses an embedded Tomcat container.

Second, it's customizable

An opinionated framework isn't much good if you can't change its mind. You can easily customize a Spring Boot application to your liking, either in the initial configuration or later in the development cycle.

For example, if you prefer Maven, then you can easily make <dependency> change(s) in your POM file to replace the Spring Boot default value. You'll do this later in the tutorial.

Get started using Spring Boot

Starters

Starters are a big part of the magic of Spring Boot, used to limit the amount of manual dependency configuration that you have to do. If you're going to use Spring Boot effectively, you should know about starters.

A *starter* is essentially a set of dependencies (such as a Maven POM) that are specific to the type of application the starter represents.

All starters use the naming convention: `spring-boot-starter-XYZ`, where XYZ is the type of application you want to build. Here are some popular Spring Boot starters:

- `spring-boot-starter-`**`web`** is used to build RESTful web services using Spring MVC and Tomcat as the embedded application container.
- `spring-boot-starter-`**`jersey`** is an alternative to `spring-boot-starter-web` that uses Apache Jersey rather than Spring MVC.
- `spring-boot-starter-`**`jdbc`** is used for JDBC connection pooling. It's based on Tomcat's JDBC connection-pool implementation.

The Spring Boot starter reference page lists **lots** more starters. Check it out to see the POM and dependencies for each starter.

Auto-configuration

If you let it, Spring Boot will use its `@EnableAutoConfiguration` annotation to automatically configure your application. Auto-configuration is based on the JARS in your classpath and how you've defined your beans:

Take a look: Fire up your Spring Boot application with the `--debug` option and an auto-configuration report will be generated to the console.

- Spring Boot uses the JARs you have specified to be present in the `CLASSPATH` to form an opinion about how to configure certain automatic behavior. For example, if you have the H2 database JAR in your classpath and have configured no other `DataSource` beans, then your application will be automatically configured with an in-memory database.
- Spring Boot uses the way you define beans to determine how to automatically configure itself. For example, if you annotate your JPA beans with `@Entity`, then Spring Boot will automatically configure JPA such that you do not need a `persistence.xml` file.

The Spring Boot über jar

Spring Boot aims to help developers create applications that "just run." To that end, it packages your application and its dependencies into a single, executable JAR. To run your application, you start Java like this:

```
$ java -jar PATH_TO_EXECUTABLE_JAR/executableJar.jar
```

The Spring Boot über JAR isn't a new concept. Because Java doesn't provide a standard way to load nested JARs, developers have been using tools like the Apache Maven Shade plugin to build "shaded" JARs for years. A shaded JAR simply contains the `.class` files from all of the application's dependent JARs. But as the complexity of your application grows and dependencies increase, shaded JARs can suffer from two issues:

1. Name collisions, where two classes in different JARs have the same name.
2. Dependency version issues, where two JARs use different versions of the same dependency.

Spring Boot resolves these issues by defining a special JAR file layout where the JARs themselves are nested within the über JAR. Spring tool support (for example, the `spring-boot-`

`maven` plugin) then builds the executable über JAR to follow that layout (not just unpacking and repackaging `.class` files, as with a shaded JAR). When you run the executable JAR, Spring Boot uses a special class loader to handle loading the classes within the nested JARs.

Say Hello, World!

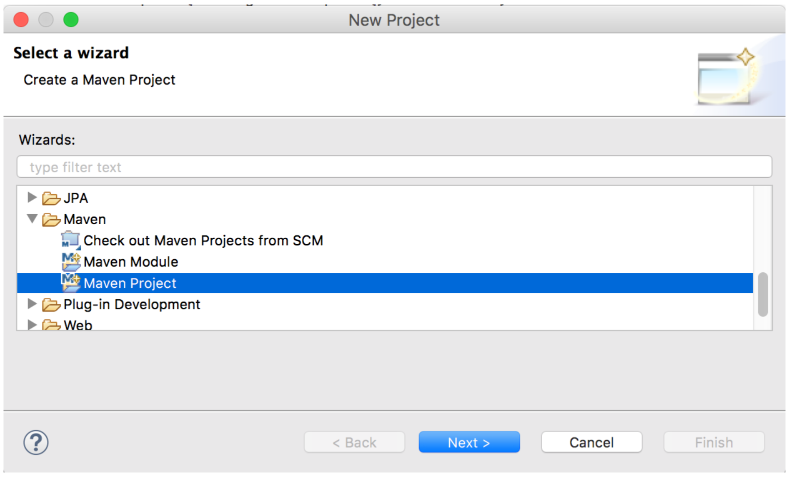
Now you're ready to start working directly with Spring Boot. Examples in this section are based on a simple application called HelloSpringBoot. I encourage you to work through the application development example with me, but if you want to jump right in you can download the application code from Github.

Let's dive right in and create a new Maven project!

1. Create the Maven project

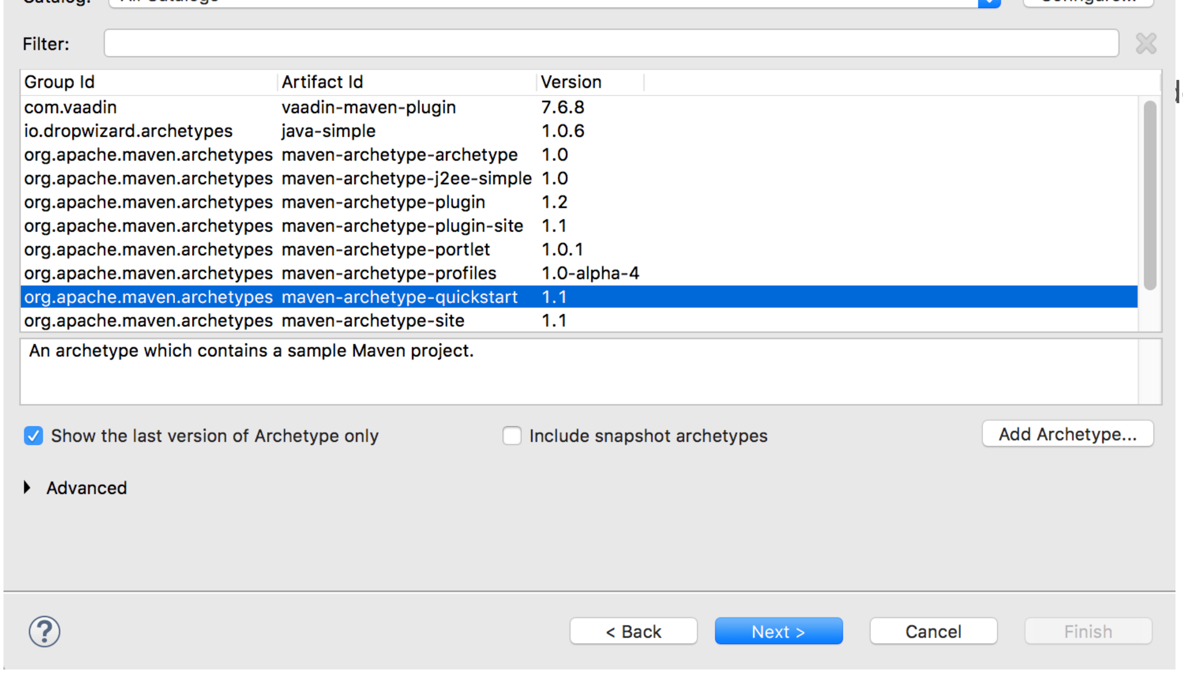
In Eclipse, go to **File > New Project** and select **Maven > Maven Project**, as shown in Figure 1.

Figure 1. Selecting a Maven project



Click Next, then click Next again on the dialog that follows (not shown).

You'll be asked to choose the archetype of your new Maven project. Select **maven-archetype-quickstart**, as shown in Figure 2.



Filter:

Group Id	Artifact Id	Version
com.vaadin	vaadin-maven-plugin	7.6.8
io.dropwizard.archetypes	java-simple	1.0.6
org.apache.maven.archetypes	maven-archetype-archetype	1.0
org.apache.maven.archetypes	maven-archetype-j2ee-simple	1.0
org.apache.maven.archetypes	maven-archetype-plugin	1.2
org.apache.maven.archetypes	maven-archetype-plugin-site	1.1
org.apache.maven.archetypes	maven-archetype-portlet	1.0.1
org.apache.maven.archetypes	maven-archetype-profiles	1.0-alpha-4
org.apache.maven.archetypes	maven-archetype-quickstart	1.1
org.apache.maven.archetypes	maven-archetype-site	1.1

An archetype which contains a sample Maven project.

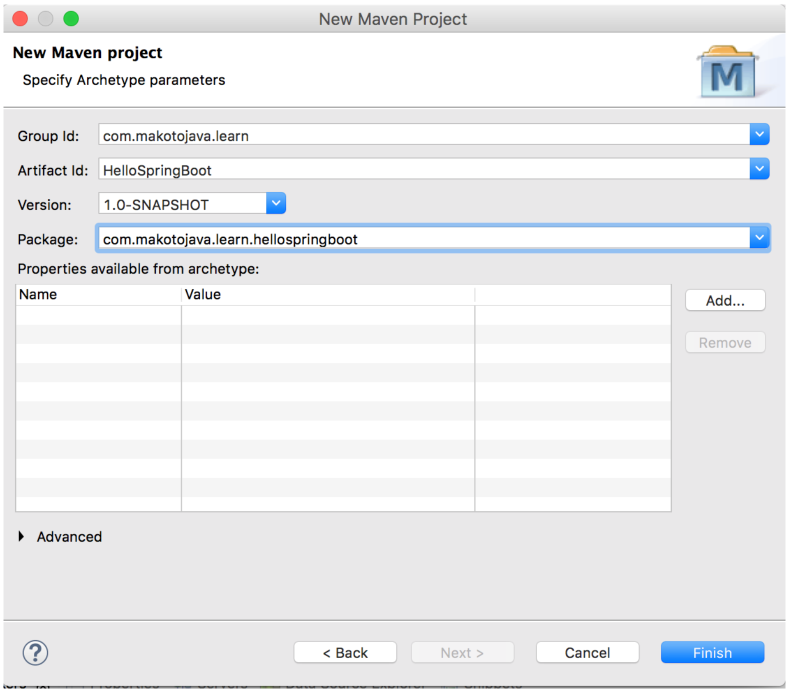
[✓] Show the last version of Archetype only

[ ] Include snapshot archetypes

Click Next.

Finally, enter the artifact settings as shown in Figure 3.

Figure 3. Selecting the Maven artifact settings


New Maven project

Specify Archetype parameters

Group Id:	com.makotojava.learn
Artifact Id:	HelloSpringBoot
Version:	1.0-SNAPSHOT
Package:	com.makotojava.learn.hellospringboot

Properties available from archetype:

I'm using the following settings for the HelloSpringBoot application:

- Group Id: `com.makotojava.learn`
- Artifact Id: `HelloSpringBoot`
- Version: `1.0-SNAPSHOT`
- Package: `com.makotojava.learn.hellospringboot`

Click Finish to create the project.

Now open `App.java` in Eclipse and replace its **entire contents** with the following:

```
        package com.makotojava.learn.hellospringboot;

        import java.util.Arrays;
        import org.slf4j.Logger;
        import org.slf4j.LoggerFactory;
        import org.springframework.boot.CommandLineRunner;
        import org.springframework.boot.SpringApplication;
        import org.springframework.boot.autoconfigure.SpringBootApplication;
        import org.springframework.context.ApplicationContext;
        import org.springframework.context.annotation.Bean;

        @SpringBootApplication
        public class App {

          private static final Logger log = LoggerFactory.getLogger(App.class);

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

          @Bean
          public CommandLineRunner commandLineRunner(ApplicationContext ctx) {
            return args -> {

              log.debug("Let's inspect the beans provided by Spring Boot:");

              String[] beanNames = ctx.getBeanDefinitionNames();
              Arrays.sort(beanNames);
              for (String beanName : beanNames) {
                log.debug(beanName);
              }

            };
          }
        }
```

Then create a new class called `HelloRestController` in the same package as `App` that looks like this:

```
        package com.makotojava.learn.hellospringboot;

        import org.springframework.web.bind.annotation.RequestMapping;
        import org.springframework.web.bind.annotation.RestController;

        @RestController
        public class HelloRestController {

          @RequestMapping("/hello")
          public String hello() {
            return "Hello. All your base are belong to us.";
          }
        }
```

2. Create the POM

Modify the POM created by the New Project wizard so it looks like Listing 1.

Listing 1. The POM file for HelloSpringBoot

```
                <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.makotojava.learn</groupId>
```

```
    <artifactId>HelloSpringBoot</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>
    <name>HelloSpringBoot</name>
    <url>http://maven.apache.org</url>
    <parent>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-starter-parent</artifactId>
     <version>1.5.2.RELEASE</version>
    </parent>
    <dependencies>
     <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
     </dependency>
    </dependencies>
    <properties>
     <java.version>1.8</java.version>
    </properties>
    <build>
     <plugins>
      <plugin>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
     </plugins>
    </build>
</project>
```

Note the highlighted lines in Listing 1:

Lines 10 through 14 show the `<parent>` element, which specifies the Spring Boot parent POM and contains definitions for common components. You don't have to manually configure these.

Lines 16 through 19 show the `<dependency>` on the `spring-boot-starter-web` Spring Boot starter. This tells Spring Boot that the application is a web application. Spring Boot will form its opinions accordingly.

Lines 25 through 30 tell Maven to use the `spring-boot-maven-plugin` plugin to generate the Spring Boot application.

That's not a lot of configuration, is it? Notice there is no XML. We'll use Java annotations for the remaining configuration.

More about Spring Boot's opinions

Before we go further, I want to talk a little more about Spring Boot's *opinions*. Namely, I think it's important to explain how Spring Boot uses a starter like `spring-boot-starter-web` to form its configuration opinions.

The example application, HelloSpringBoot, uses Spring Boot's web application starter, `spring-boot-starter-web`. Based on this starter, Spring Boot has formed the following opinions about the application:

- Tomcat embedded web server container
- Hibernate for Object-Relational Mapping (ORM)

- Apache Jackson for JSON binding
- Spring MVC for the REST framework

Talk about opinionated! But in Spring Boot's defense, these are the most popular web application defaults—at least I know I use them all the time.

But remember how I said Spring Boot is customizable? If you want to use a different technology stack, you can easily override Spring Boot's defaults.

We'll walk through a simple customization next.

Lose the `<parent>`

What if if you already have a `<parent>` element in your POM, or what if just don't want to use it? Will Spring Boot still work?

Yes it will, but you have to do two things:

1. Manually add the dependencies (including the versions)
2. Add a configuration snippet to your `spring-boot-maven-plugin`, as shown in Listing 2:

Listing 2. Specifying the `repackage` goal when not using the <parent> POM element

```
<build>
 <plugins>
  <plugin>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-maven-plugin</artifactId>
   <version>1.5.2.RELEASE</version>
   <executions>
    <execution>
     <goals>
      <goal>repackage</goal>
     </goals>
    </execution>
   </executions>
  </plugin>
 </plugins>
</build>
```

Check the dependencies: If you ever need to see exactly what dependencies are being pulled in by the Spring Boot starter you're using, visit the Spring Boot starter reference page.

It's important to note that the <parent> element does a lot of cool Maven magic, so if you have a good reason for not using it, proceed with caution. Make sure to add a repackage goal execution to the `spring-boot-maven-plugin`as explained here.

The project is configured and customized. Now it's time to build the executable.

3. Build the executable JAR

You have two options for building the executable JAR using Maven:

1. Run the Maven build in Eclipse

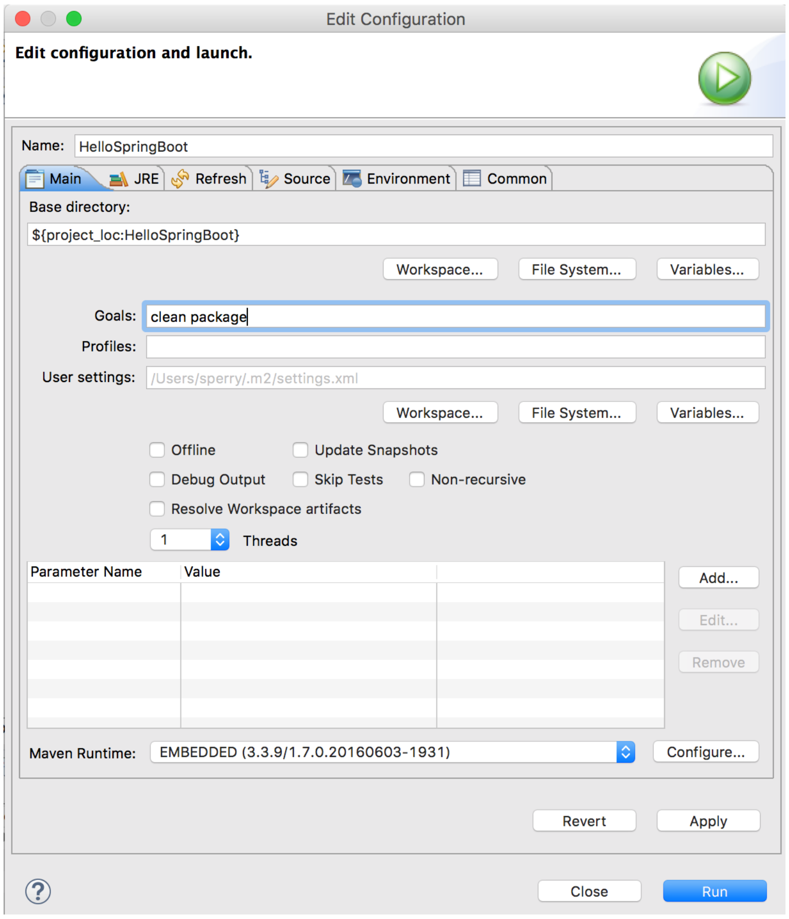
2. Run the Maven build from the command line

I'll show you how to do both.

Build in Eclipse

To run the Maven build in Eclipse, right-click the POM file and choose **Run As > Maven Build**. In the Goals text field, enter `clean` and `package`, then click the Run button.

Figure 4. Build the executable JAR in Eclipse



Name: HelloSpringBoot
Base directory: ${project_loc:HelloSpringBoot}
Goals: clean package
Profiles:
User settings: /Users/sperry/.m2/settings.xml
Threads: 1
Maven Runtime: EMBEDDED (3.3.9/1.7.0.20160603-1931)

You should see a message in the console view indicating a successful build:

```
.
.
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2.440 s
[INFO] Finished at: 2017-04-16T10:17:21-05:00
[INFO] Final Memory: 30M/331M
[INFO] ------------------------------------------------------------------------
```

Build from the command line

To run the Maven build from the command line, open a Mac terminal window or a Windows command prompt, navigate to the `HelloSpringBoot` project directory, and execute the command:

```
mvn clean package
```

You should see a message in the terminal window or command prompt indicating a successful build.

```
                $ cd HelloSpringBoot
                $ pwd
                /Users/sperry/home/HelloSpringBoot
                $ mvn clean package
                .
                .
                [INFO] ------------------------------------------------------------------------
                [INFO] BUILD SUCCESS
                [INFO] ------------------------------------------------------------------------
                [INFO] Total time: 2.440 s
                [INFO] Finished at: 2017-04-16T10:17:21-05:00
                [INFO] Final Memory: 30M/331M
                [INFO] ------------------------------------------------------------------------
                $
```

Now you're ready to run the executable JAR.

4. Run the executable JAR

To run the executable JAR you've just created, open a Mac terminal window or a Windows command prompt, navigate to the `HelloSpringBoot` project folder, and execute:

```
java -jar target/HelloSpringBoot-1.0-SNAPSHOT.jar
```

where `target` is the default output directory of the build. If you have configured it differently, please make the appropriate substitution in the command above.

The output from Spring Boot contains a text-based "Splash Screen" (lines 2 to 7) along with other output, similar to the listing below. I've showed just a few lines, to give you an idea of what you should see when you run the app:

```
$ java -jar target/HelloSpringBoot-1.0-SNAPSHOT.jar
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.5.2.RELEASE)
2017-04-15 17:46:12.919  INFO 20096 --- [           main] c.makotojava.learn.hellospringboot.App   : Starting
 App v1.0-SNAPSHOT on Ix.local with PID 20096 (/Users/sperry/home/projects/learn/HelloSpringBoot/target/
HelloSpringBoot-1.0-SNAPSHOT.jar started by sperry in /Users/sperry/home/projects/learn/HelloSpringBoot)
2017-04-15 17:46:12.924 DEBUG 20096 --- [           main] c.makotojava.learn.hellospringboot.App   : Running
 with Spring Boot v1.5.2.RELEASE, Spring v4.3.7.RELEASE
.
.
2017-04-15 17:46:15.221  INFO 20096 --- [           main] c.makotojava.learn.hellospringboot.App   : Started
 App in 17.677 seconds (JVM running for 18.555)
```

If the application starts successfully, the last line of output from Spring Boot will contain the words "Started App" (line 13). Now you're ready to exercise your application, which you'll do next.

5. Exercise the app

You can execute HelloSpringBoot's single REST method by opening a browser and hitting the following URL:

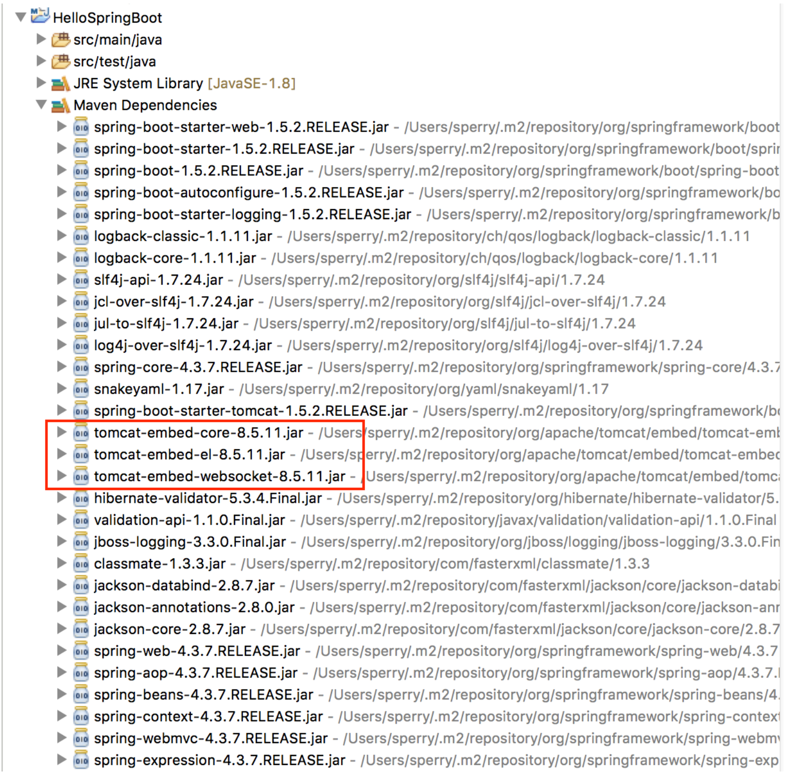
```
http://localhost:8080/hello
```

If you see the text "Hello, All your base are belong to us" (an homage to the video game Zero Wing), then you know the application works!

Changing Spring Boot's opinions

Spring Boot's opinions are based on the contents of the POM, including the Spring Boot starter you specify when you initially configure your application. After forming an opinion about the type of application you intend to build, Spring Boot delivers a set of Maven dependencies. Figure 4 shows some of the Maven dependencies Spring Boot has set up in Eclipse, based on the POM contents and starter specified for the HelloSpringBoot application:

Figure 6. Initial dependencies for HelloSpringBoot



```
▼ HelloSpringBoot
  ▶ src/main/java
  ▶ src/test/java
  ▶ JRE System Library [JavaSE-1.8]
  ▼ Maven Dependencies
    ▶ spring-boot-starter-web-1.5.2.RELEASE.jar - /Users/sperry/.m2/repository/org/springframework/boot
    ▶ spring-boot-starter-1.5.2.RELEASE.jar - /Users/sperry/.m2/repository/org/springframework/boot/sprir
    ▶ spring-boot-1.5.2.RELEASE.jar - /Users/sperry/.m2/repository/org/springframework/boot/spring-boot
    ▶ spring-boot-autoconfigure-1.5.2.RELEASE.jar - /Users/sperry/.m2/repository/org/springframework/bo
    ▶ spring-boot-starter-logging-1.5.2.RELEASE.jar - /Users/sperry/.m2/repository/org/springframework/b
    ▶ logback-classic-1.1.11.jar - /Users/sperry/.m2/repository/ch/qos/logback/logback-classic/1.1.11
    ▶ logback-core-1.1.11.jar - /Users/sperry/.m2/repository/ch/qos/logback/logback-core/1.1.11
    ▶ slf4j-api-1.7.24.jar - /Users/sperry/.m2/repository/org/slf4j/slf4j-api/1.7.24
    ▶ jcl-over-slf4j-1.7.24.jar - /Users/sperry/.m2/repository/org/slf4j/jcl-over-slf4j/1.7.24
    ▶ jul-to-slf4j-1.7.24.jar - /Users/sperry/.m2/repository/org/slf4j/jul-to-slf4j/1.7.24
    ▶ log4j-over-slf4j-1.7.24.jar - /Users/sperry/.m2/repository/org/slf4j/log4j-over-slf4j/1.7.24
    ▶ spring-core-4.3.7.RELEASE.jar - /Users/sperry/.m2/repository/org/springframework/spring-core/4.3.7
    ▶ snakeyaml-1.17.jar - /Users/sperry/.m2/repository/org/yaml/snakeyaml/1.17
    ▶ spring-boot-starter-tomcat-1.5.2.RELEASE.jar - /Users/sperry/.m2/repository/org/springframework/bo
    ▶ tomcat-embed-core-8.5.11.jar - /Users/sperry/.m2/repository/org/apache/tomcat/embed/tomcat-emt
    ▶ tomcat-embed-el-8.5.11.jar - /Users/sperry/.m2/repository/org/apache/tomcat/embed/tomcat-embec
    ▶ tomcat-embed-websocket-8.5.11.jar - /Users/sperry/.m2/repository/org/apache/tomcat/embed/tomca
    ▶ hibernate-validator-5.3.4.Final.jar - /Users/sperry/.m2/repository/org/hibernate/hibernate-validator/5.
    ▶ validation-api-1.1.0.Final.jar - /Users/sperry/.m2/repository/javax/validation/validation-api/1.1.0.Final
    ▶ jboss-logging-3.3.0.Final.jar - /Users/sperry/.m2/repository/org/jboss/logging/jboss-logging/3.3.0.Fin
    ▶ classmate-1.3.3.jar - /Users/sperry/.m2/repository/com/fasterxml/classmate/1.3.3
    ▶ jackson-databind-2.8.7.jar - /Users/sperry/.m2/repository/com/fasterxml/jackson/core/jackson-databi
    ▶ jackson-annotations-2.8.0.jar - /Users/sperry/.m2/repository/com/fasterxml/jackson/core/jackson-anr
    ▶ jackson-core-2.8.7.jar - /Users/sperry/.m2/repository/com/fasterxml/jackson/core/jackson-core/2.8.7
    ▶ spring-web-4.3.7.RELEASE.jar - /Users/sperry/.m2/repository/org/springframework/spring-web/4.3.7
    ▶ spring-aop-4.3.7.RELEASE.jar - /Users/sperry/.m2/repository/org/springframework/spring-aop/4.3.7.
    ▶ spring-beans-4.3.7.RELEASE.jar - /Users/sperry/.m2/repository/org/springframework/spring-beans/4.
    ▶ spring-context-4.3.7.RELEASE.jar - /Users/sperry/.m2/repository/org/springframework/spring-context
    ▶ spring-webmvc-4.3.7.RELEASE.jar - /Users/sperry/.m2/repository/org/springframework/spring-webmv
    ▶ spring-expression-4.3.7.RELEASE.jar - /Users/sperry/.m2/repository/org/springframework/spring-exp
```

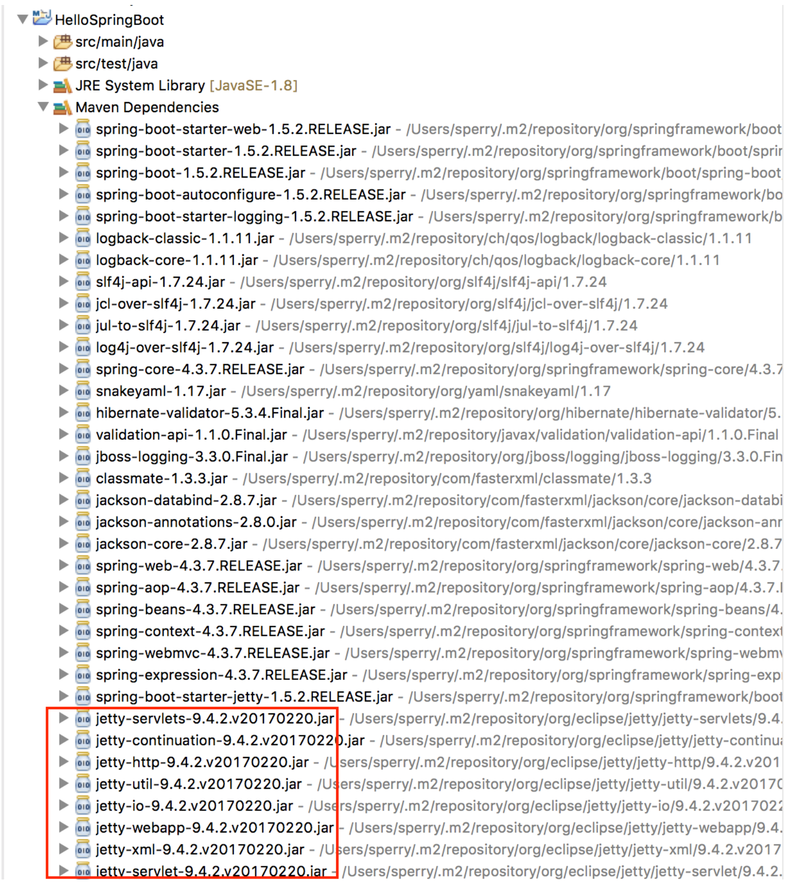
Note that Tomcat is the default embedded web server container. Now let's suppose that instead of Tomcat you want to use Jetty. All you need to do is change the `<dependencies>` section in the POM (just paste lines 5 through 15 from Listing 3 over line 19 from Listing 1):

Listing 3. POM change to use Jetty instead of Tomcat

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

Notice below that the Maven dependencies for Tomcat are gone (thanks to lines 5 through 10 in Listing 3), replaced with dependencies for Jetty (lines 12 through15).

Figure 7. Customized dependencies for HelloSpringBoot



```
▼ HelloSpringBoot
  ▶ src/main/java
  ▶ src/test/java
  ▶ JRE System Library [JavaSE-1.8]
  ▼ Maven Dependencies
    ▶ spring-boot-starter-web-1.5.2.RELEASE.jar - /Users/sperry/.m2/repository/org/springframework/boot
    ▶ spring-boot-starter-1.5.2.RELEASE.jar - /Users/sperry/.m2/repository/org/springframework/boot/sprir
    ▶ spring-boot-1.5.2.RELEASE.jar - /Users/sperry/.m2/repository/org/springframework/boot/spring-boot
    ▶ spring-boot-autoconfigure-1.5.2.RELEASE.jar - /Users/sperry/.m2/repository/org/springframework/bo
    ▶ spring-boot-starter-logging-1.5.2.RELEASE.jar - /Users/sperry/.m2/repository/org/springframework/b
    ▶ logback-classic-1.1.11.jar - /Users/sperry/.m2/repository/ch/qos/logback/logback-classic/1.1.11
    ▶ logback-core-1.1.11.jar - /Users/sperry/.m2/repository/ch/qos/logback/logback-core/1.1.11
    ▶ slf4j-api-1.7.24.jar - /Users/sperry/.m2/repository/org/slf4j/slf4j-api/1.7.24
    ▶ jcl-over-slf4j-1.7.24.jar - /Users/sperry/.m2/repository/org/slf4j/jcl-over-slf4j/1.7.24
    ▶ jul-to-slf4j-1.7.24.jar - /Users/sperry/.m2/repository/org/slf4j/jul-to-slf4j/1.7.24
    ▶ log4j-over-slf4j-1.7.24.jar - /Users/sperry/.m2/repository/org/slf4j/log4j-over-slf4j/1.7.24
    ▶ spring-core-4.3.7.RELEASE.jar - /Users/sperry/.m2/repository/org/springframework/spring-core/4.3.7
    ▶ snakeyaml-1.17.jar - /Users/sperry/.m2/repository/org/yaml/snakeyaml/1.17
    ▶ hibernate-validator-5.3.4.Final.jar - /Users/sperry/.m2/repository/org/hibernate/hibernate-validator/5.
    ▶ validation-api-1.1.0.Final.jar - /Users/sperry/.m2/repository/javax/validation/validation-api/1.1.0.Final
    ▶ jboss-logging-3.3.0.Final.jar - /Users/sperry/.m2/repository/org/jboss/logging/jboss-logging/3.3.0.Fin
    ▶ classmate-1.3.3.jar - /Users/sperry/.m2/repository/com/fasterxml/classmate/1.3.3
    ▶ jackson-databind-2.8.7.jar - /Users/sperry/.m2/repository/com/fasterxml/jackson/core/jackson-databi
    ▶ jackson-annotations-2.8.0.jar - /Users/sperry/.m2/repository/com/fasterxml/jackson/core/jackson-anr
    ▶ jackson-core-2.8.7.jar - /Users/sperry/.m2/repository/com/fasterxml/jackson/core/jackson-core/2.8.7
    ▶ spring-web-4.3.7.RELEASE.jar - /Users/sperry/.m2/repository/org/springframework/spring-web/4.3.7.
    ▶ spring-aop-4.3.7.RELEASE.jar - /Users/sperry/.m2/repository/org/springframework/spring-aop/4.3.7.
    ▶ spring-beans-4.3.7.RELEASE.jar - /Users/sperry/.m2/repository/org/springframework/spring-beans/4.
    ▶ spring-context-4.3.7.RELEASE.jar - /Users/sperry/.m2/repository/org/springframework/spring-context
    ▶ spring-webmvc-4.3.7.RELEASE.jar - /Users/sperry/.m2/repository/org/springframework/spring-webm
    ▶ spring-expression-4.3.7.RELEASE.jar - /Users/sperry/.m2/repository/org/springframework/spring-exp
    ▶ spring-boot-starter-jetty-1.5.2.RELEASE.jar - /Users/sperry/.m2/repository/org/springframework/boot
    ▶ jetty-servlets-9.4.2.v20170220.jar - /Users/sperry/.m2/repository/org/eclipse/jetty/jetty-servlets/9.4.
    ▶ jetty-continuation-9.4.2.v20170220.jar - /Users/sperry/.m2/repository/org/eclipse/jetty/jetty-continu
    ▶ jetty-http-9.4.2.v20170220.jar - /Users/sperry/.m2/repository/org/eclipse/jetty/jetty-http/9.4.2.v201
    ▶ jetty-util-9.4.2.v20170220.jar - /Users/sperry/.m2/repository/org/eclipse/jetty/jetty-util/9.4.2.v2017C
    ▶ jetty-io-9.4.2.v20170220.jar - /Users/sperry/.m2/repository/org/eclipse/jetty/jetty-io/9.4.2.v2017022
    ▶ jetty-webapp-9.4.2.v20170220.jar - /Users/sperry/.m2/repository/org/eclipse/jetty/jetty-webapp/9.4.
    ▶ jetty-xml-9.4.2.v20170220.jar - /Users/sperry/.m2/repository/org/eclipse/jetty/jetty-xml/9.4.2.v2017
    ▶ jetty-servlet-9.4.2.v20170220.jar - /Users/sperry/.m2/repository/org/eclipse/jetty/jetty-servlet/9.4.2.
```

Actually, there are more Jetty dependencies than would fit in a single screenshot, but they are all there, and the Tomcat dependencies are gone. Try it and see for yourself!

Say Hello, Galaxy!

Simple examples are great, but Spring Boot is capable of so much more! In this section, I'll show you how to put Spring Boot through its paces with a Spring MVC RESTful web application. The first thing to do is set up the new example application, SpringBootDemo.

SpringBootDemo

SpringBootDemo is a Spring Boot wrapper around a simple Spring-based POJO application called oDoT. (For ToDo backwards ... get it?) The idea is to walk through the process of developing an application that is more complex than a simple Hello, World. You'll also learn how to wrap an existing application with Spring Boot.

You'll do three things to setup and run SpringBootDemo:

1. Get the code from GitHub.
2. Build and run the executable JAR.
3. Access the application through SoapUI.

I've created a video to help guide you through each step of this process. Feel free to start the video now.

To view this video, **SpringBootDemo walk through** , please access the online version of the article. If this article is in the developerWorks archives, the video is no longer accessible.

1. Get the code

To start, you will need to clone two projects from their GitHub repositories. The first, called odotCore, contains the application's business logic, which is written as a Spring-based POJO application. The other, called SpringBootDemo, is a Spring Boot application wrapper around odotCore.

To clone the odotCore repo, open a Mac terminal window or a Windows command prompt, navigate to the root directory under which you want the code to reside, and execute the command:

```
git clone https://github.com/makotogo/odotCore
```

To clone the SpringBootDemo repo, execute the command:

```
git clone https://github.com/makotogo/SpringBootDemo
```

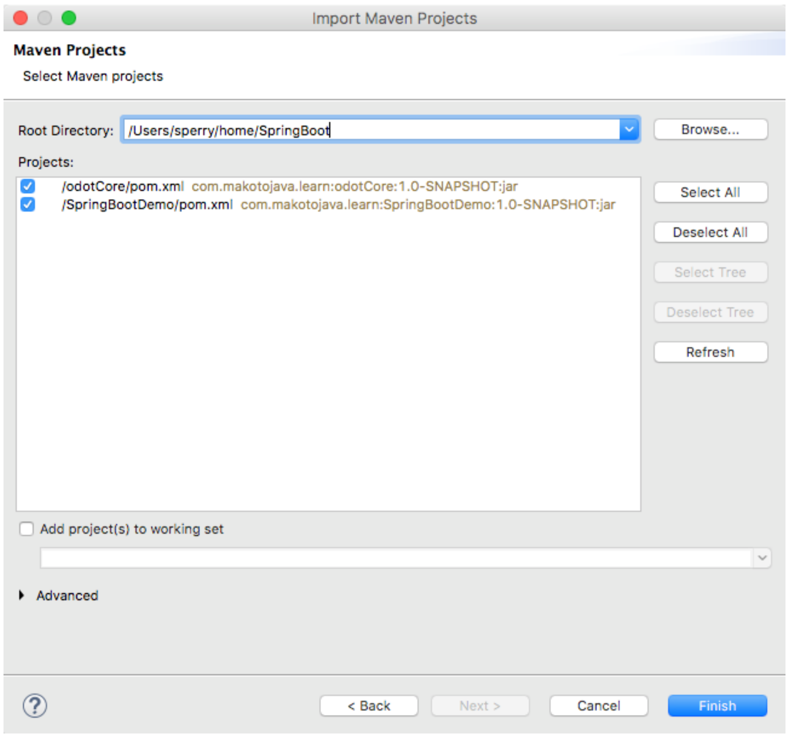
Note that the two projects are immediately subordinate to the application's root directory. Next you'll import the code to your workspace.

2. Import the code into Eclipse

Go to **File > Import...** and choose **Maven > Existing Maven Projects**.

In the next dialog, use the Browse button to navigate to the root directory. Both of the projects cloned in the previous step should appear in the dialog, as shown here:

Figure 8. Import Maven Projects



Import Maven Projects

Maven Projects

Select Maven projects

Root Directory: /Users/sperry/home/SpringBoot

Projects:

- [✓] /odotCore/pom.xml com.makotojava.learn:odotCore:1.0-SNAPSHOT:jar
- [✓] /SpringBootDemo/pom.xml com.makotojava.learn:SpringBootDemo:1.0-SNAPSHOT:jar

Click Finish to import the projects into your Eclipse workspace. Next you'll build the executable JAR.

3. Build the executable JAR

Building SpringBootDemo requires you to build both the odotCore and SpringBootDemo projects. It is possible to build the projects from the command line, as you saw with the HelloSpringBoot application. In this case, I'll walk you through using Eclipse.

In Eclipse, right-click on the odotCore project. Choose **Run As > Maven Build** and specify the `clean` and `install` goals. The `install` goal will install the `odotCore-1.0-SNAPSHOT.jar` JAR file into your local Maven repository. From here, it will be available to pull in as a dependency when you run the SpringBootDemo Maven build.

After the `odotCore` Maven build runs successfully, right-click on the SpringBootDemo project, choose **Run As > Maven Build** and specify the `clean` and `package` goals.

Note: The odotCore project contains several unit tests. While I recommend never (ever) skipping unit tests, you can setup the Run Configuration that builds the odotCore project in Eclipse to skip the tests (there's a check box for that on the Run Configuration dialog).

After the SpringBootDemo build has run successfully, you can run the SpringBootDemo über JAR from the command line.

4. Run the executable JAR

From a Mac terminal window or Windows command prompt, navigate to the SpringBootDemo directory. Assuming the build's output directory is called `target` (which is the default), execute the following command:

```
java -jar target/SpringBootDemo-1.0-SNAPSHOT.jar
```

Now stand back in awe as Spring Boot runs the application. When you see the text "App Started" you're ready to exercise the application.

5. Exercise the app

As a quick smoke test, to make sure your application is working correctly, open a browser window and enter the following URL:

```
http://localhost:8080/CategoryRestService/FindAll
```

This accesses the `FindAll` method of the `CategoryRestService` and returns all of the `Category` objects in the database in JSON format.

You can also exercise the app through SoapUI. I won't demonstrate how to do that here, but I do guide you through the process in the video for this tutorial.

Table 1 shows the services and methods within each service for the SpringBootDemo.

Table 1. oDoT (SpringBootDemo) services and methods

Service	Method	HTTP method	Example URL @ http://localhost:8080	
Category	FindAll	GET	/CategoryRestService/FindAll	Finds all Category objects in the DB.
Category	FindById	GET	/CategoryRestService/FindbyId/1	Finds Category by ID value 1.
Category	FindById	GET	/CategoryRestService/FindbyName/MY_CATEGORY	Finds Category by name value "MY_CATEGORY".
Category	Add	PUT	/CategoryRestService/Add	Adds the specified Category (as JSON payload in request body) to the DB. Returns: Category object that was added (as JSON in response body).
Category	Update	POST	/CategoryRestService/Update	Updates the specified Category (as JSON payload in request body) to the DB. Returns: String message indicating status of the update.
Category	Delete	DELETE	/CategoryRestService/Delete	Deletes the specified Category (as JSON payload in request body) to the DB. Returns: String message indicating status of the delete.
Item	FindAll	GET	/ItemRestService/FindAll	Finds all Category objects in the DB.
Item	FindById	GET	/ItemRestService/FindbyId/1	Finds Category by ID value 1.
Item	FindById	GET	/ItemRestService/FindbyName/TODO_ITEM_1	Finds Item by name value "TODO_ITEM_1".
Item	Add	PUT	/ItemRestService/Add	Adds the specified Item (as JSON payload in request body) to the DB. Returns: Item object that was added (as JSON in response body).
Item	Update	POST	/ItemRestService/Update	Updates the specified Item (as JSON payload in request body) to the DB. Returns: String message indicating status of the update.
Item	Delete	DELETE	/ItemRestService/Delete	Deletes the specified Item (as JSON payload in request body) to the DB.

				Returns: String message indicating status of the delete.

I recommend you study the code, play around with it, and get a better feeling for how Spring Boot works.

Conclusion

In this tutorial I introduced you to the problems Spring Boot solves and a little about how it works. Then I walked you through setting up and running a simple Spring Boot application called HelloSpringBoot. Finally, I showed you how to build and exercise a Spring MVC RESTful web services application using Spring Boot.

So where you do you go from here?

Sign up for developerWorks Premium

Related topics

- What is Spring Boot?
- How to deploy Spring Boot application in IBM Liberty and WAS 8.5