

Learning Spring Boot

Greg L. Turnquist



Chapter No. 2 "Quick Start with Java"

In this package, you will find:

The author's biography

A preview chapter from the book, Chapter no.2 "Quick Start with Java"

A synopsis of the book's content

Information on where to buy this book

About the Author

Greg L. Turnquist has developed software professionally since 1997. From 2002 to 2010, he was part of the senior software team that worked on Harris' \$3.5 billion FAA telco program, architecting mission-critical enterprise apps while managing a software team. He provided after-hours support to a nation-wide telco system and is no stranger to midnight failures and software triages. In 2010, he joined the SpringSource division of VMware, which was spun off into Pivotal in 2013.

As a test-bitten script junky, Java geek, and JavaScript Padawan, he is a member of the Spring Data team as well as the mobile-oriented Allspark team. He has made key contributions to Spring Boot and Spring Data REST while also serving as Getting Started Guides, editor-at-large for <http://spring.io/>. He has migrated Spring Data release train's entire reference docs to AsciiDoctor in a week. He has also contributed to multiple Spring portfolio projects.

For More Information:

www.packtpub.com/application-development/learning-spring-boot

He has worked with Java, Spring, Spring Security, AspectJ, and Jython technologies and has also developed sophisticated scripts for *nix and Windows platforms. As a wiki evangelist, he has also deployed a LAMP-based wiki website that provides fingertip knowledge to users.

In 2006, Greg created the Spring Python project. The Spring Framework provided many useful features, and he wanted these features to be available when he was working with Python. He has written *Python Testing Cookbook* and *Spring Python 1.1* for Packt Publishing.

He has completed a Master's degree in Computer Engineering at Auburn University and lives in the United States with his family.

For More Information:

www.packtpub.com/application-development/learning-spring-boot

Learning Spring Boot

"Very impressed with @springboot so far, 10 mins to get a REST service up and running, now to add MongoDB. No black magic under the covers!"

— Graham Rivers-Brown, <https://twitter.com/grahamrb/status/500422704499945472>

Back in 2012, a couple of Spring developers stepped back from the current state of Java development and asked some questions, "Developers with Spring MVC on the classpath probably want to use it. How can we make this easier?" "How can we make Spring more accessible to new developers?" "Why does a computer student have to learn about build systems, web.xml files, and all the other steps to simply display "Hello, World" on a web page?"

By approaching the topic of simplifying development without sacrificing the power of Spring, they coded several powerful features. At the SpringOne conference in 2013, they unveiled Spring Boot with its ability to auto-configure Spring Beans, configure things with simple property management, and run apps inside an embedded Tomcat container bundled inside a runnable JAR file. They also showed Spring Boot's opinionated approach to pick which Spring beans were configured based on classpath settings among other factors.

The response was incredible. The session by Phil Webb and Dave Syer witnessed record attendance along with lots of follow-up questions. (I know because I was there.) People were discussing it in the hallways between talks. The ensuing stream of blog articles after the conference from the Spring community was relentless. Proof of its success and staying power was further evidenced at the 2014 SpringOne conference a year later. Spring Boot had woven itself into almost every Java-based demo as a new lingua franca among Spring developers.

The keynote presented by Andy Glover, a Netflix engineer who had once developed Java but left to write Ruby on Rails, explained the reasons why he returned to Java. Spring Boot made Java fun again! Furthermore, when a contingent of over 20 Spring developers from the conference visited the Java Metroplex Users Group in Dallas the same week, there was a lot of excitement. Lots of questions were fired off to the Spring team with many about Spring Boot, including "When will there be a book about Spring Boot?"

I hope you enjoy this experience.

For More Information:

www.packtpub.com/application-development/learning-spring-boot

What This Book Covers

Chapter 1, Quick Start with Groovy, explains how to rapidly craft a Spring MVC app that runs inside an embedded Tomcat container using just a few lines of Groovy and no build file. You will also learn how to plug in jQuery, web templates, and production-grade metrics and health checks.

Chapter 2, Quick Start with Java, explains how to rapidly create a Spring MVC app with Java that connects to GitHub and scans for open issues using Spring Social GitHub. Then create a mobile frontend and deploy it to the cloud.

Chapter 3, Debugging and Managing Your App, explains how to create a JMS-based publisher/subscriber app with embedded ActiveMQ that simulates ops center monitoring. You will learn how Spring Boot auto-configures things as well as how to override its opinion with your own. Also, you can add customized health checks and custom metrics, and reconfigure Spring Boot's default management settings.

Chapter 4, Data Access with Spring Boot, explains how to spin up a sports team app backed by a relational database using Spring Data JPA. You will see how to use Spring Boot's support of Spring profiles to have an in-memory database for development while switching to a persistent one for production. You will also discover how Spring Boot auto-configures database support. You will learn how to export the database layer with Spring Data REST as a hypermedia-based RESTful interface based on several REST standards. Finally, you will get a taste of switching to Spring Data MongoDB.

Chapter 5, Securing Your App with Spring Boot, explains how to create a fully functional sports-team roster app with Spring MVC and then secure it with Spring Security. You will also learn how to control security through both URL and method-level rules. Next, you will discover how to configure Spring Boot's embedded Tomcat servlet container to also serve things via SSL. This chapter also explains how to fine-tune your security policies to force traffic over encrypted channels to protect user data by default.

For More Information:

www.packtpub.com/application-development/learning-spring-boot

2

Quick Start with Java

"With Boot you deploy everywhere you can find a JVM basically."

– Oliver Gierke @olivergierke

In the previous chapter, we saw how quickly an application can be created with just a few lines of code. To fit the more commonly used paradigm, this chapter (and the rest of the book) will use a project with a build file and Java code instead. However, we'll still see how Spring Boot makes things quick and easy.

In this chapter, we are going to build an app that scans GitHub issues and uses Spring Boot to help guide us in reducing the complexity of integrating multiple Spring projects as well as other third-party libraries.

In this chapter, we will be:

- Using `http://start.spring.io` to create a bare bones Spring Boot project with Gradle support
- Creating a simple app that looks for open issues in multiple GitHub repositories
- Supplying GitHub credentials using Boot's über easy property support
- Learning how Boot finds templates
- Adding mobile support using Spring Mobile and jQuery Mobile
- Bundling up the application as a runnable JAR and deploying it to Cloud Foundry
- Adding production-ready support with Actuator and then writing a script to poll for usage metrics

For More Information:

www.packtpub.com/application-development/learning-spring-boot

Creating an empty project with start.spring.io

To kick things off, we need a new project. Instead of starting from absolutely nothing, Spring Boot provides a website that is used to create new projects at <http://start.spring.io>. We enter some information, pick a set of desired options, and then download either a build file or a zipped-up project.

The screenshot shows the Spring Initializr web application in a browser window. The page has a dark header with the text "SPRING INITIALIZR" and "Bootstrap your application now". Below the header, the page is divided into two main sections: "Project metadata" on the left and "Project dependencies" on the right.

Project metadata

- Group:** learningspringboot
- Artifact:** issue-manager
- Name:** Issue Manager
- Description:** Learning Spring Boot
- Package Name:** learningspringboot
- Type:** Gradle Project
- Packaging:** Jar

Project dependencies

- Core:**
 - ☐ Security
 - ☐ AOP
- Data:**
 - ☐ JDBC
 - ☐ JPA
 - ☐ MongoDB
 - ☐ Redis
 - ☐ Gemfire
 - ☐ Solr
 - ☐ Elasticsearch
- I/O:**
 - ☐ Batch
 - ☐ Integration
 - ☐ JMS
 - ☐ AMQP
- Web:**
 - ☐ Web
 - ☐ Websocket
 - ☐ WS
 - ☐ Rest Repositories
 - ☐ Mobile
- Template Engines:**
- Social:**
 - ☐ Facebook

The screen is a bit long and was cut off. The following table shows you all the settings filled in for this example. However, to see the code behind this website, visit <https://github.com/spring-io/initializr>. To whet your appetite, the site is, in fact, a Spring Boot / Groovy app using the same tools covered in the previous chapter.

As shown in the previous screenshot, we entered this information:

Field	Value
Group	learningspringboot
Artifact	issue-manager
Name	Issue Manager
Description	Learning Spring Boot
Package Name	learningspringboot
Type	Gradle Project
Packaging	Jar
Java Version	1.8
Language	Java
Project dependencies	Thymeleaf

Click on the **Generate Project** button; it downloads `starter.zip`. Let's take a peek inside the ZIP file:

```
$ unzip -l <downloaded zip file>
Archive:  starter.zip
Length      Date       Time      Name
-----
0   06-13-14 03:37   src/
0   06-13-14 03:37   src/main/
0   06-13-14 03:37   src/main/java/
0   06-13-14 03:37   src/main/java/learningspringboot/
0   06-13-14 03:37   src/main/resources/
0   06-13-14 03:37   src/test/
0   06-13-14 03:37   src/test/java/
0   06-13-14 03:37   src/test/java/learningspringboot/
1421 06-13-14 03:37   build.gradle
466 06-13-14 03:37   src/main/java/learningspringboot/Application.
java
0   06-13-14 03:37   src/main/resources/application.properties
404 06-13-14 03:37   src/test/java/learningspringboot/
ApplicationTests.java
-----
2291                                12 files
```


Let's take a look at what we have:

- A standard Gradle project layout (`src/main/java`, `src/main/resources`, `src/test/java`, and `src/test/resources`)
- The root Java package, which is `learningspringboot`
- A `build.gradle` file, which we'll look at later in this chapter
- A couple of classes that are already created: `Application.java` and `ApplicationTests.java`
- A properties file (`application.properties`) that we'll discuss in the next section



Feel free to pick to pick the build system you want. Spring Boot has equivalent support for Maven. It's also possible to use Ant, but Spring Boot has no special support for it. For reasons of space and other factors, this book will focus on Gradle and not show apps expressed in any other build system.

Before looking at the generated code, let's look at the build file:

```
// tag::plugins[]
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-
            plugin:1.1.6.RELEASE")
    }
}
// end::plugins[]

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

jar {
    baseName = 'issue-manager'
    version = '0.0.1-SNAPSHOT'
}

// tag::version[]
sourceCompatibility = 1.8
```

```

targetCompatibility = 1.8
// end::version[]

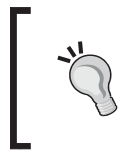
repositories {
    mavenCentral()
}

// tag::dependencies[]
dependencies {
    compile("org.springframework.boot:spring-boot-starter-
        thymeleaf")
    testCompile("org.springframework.boot:spring-boot-starter-
        test")
}
// end::dependencies[]

task wrapper(type: Wrapper) {
    gradleVersion = '2.1'
}

```

There are a quite a few parts to this file, so let's walk through them bit by bit.



You might see some comments such as `<!-- tag::x-y-z[] -->` in `build.gradle` and other files throughout this book. These are simple comments that are used to help pull in subsections for more detailed explanations and are *not* required to run any code you write.

The first important nugget at the top is this:

```

buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-
            plugin:1.1.6.RELEASE")
    }
}

```

This shows you our project, which is configured to pull down packages from mavenCentral. However, more importantly, our package is using `spring-boot-gradle-plugin`, Version 1.1.6.RELEASE. A key feature that we will see in this chapter and through the rest of this book is Spring Boot's series of predefined versions for many third-party libraries (not just Spring projects). By using this plugin, Spring Boot will set the version number for any dependency we declare that it happens to manage.

Continuing to check out our build file, we can see a list of dependencies:

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

These include the following:

- `spring-boot-starter-thymeleaf`: This pulls in dependencies that are required in order to use Thymeleaf as our view engine
- `spring-boot-starter-test`: This pulls in Spring test utilities when we are running tests

The first one matches the checkbox we picked on the form (but couldn't see directly due to the cutoff): **Thymeleaf**. The second one is included in all projects, given the popularity of automated testing in this day and age.

So, what are these quirky packages? They definitely look different than any packages we might have used in the past, which we will find out about in the next section.

Before we do that, let's look at another key setting:

```
sourceCompatibility = 1.8  
targetCompatibility = 1.8
```

This specifies that the project is using Java 8. While Spring Boot provides support as far back as Java 6, we plan to take advantage of the latest features that are out there throughout this book.

Spring Boot starters

The packages that were plugged in by `start.spring.io` are known as **Spring Boot starters**. They are virtual packages that are deployed to Maven central. Their job is to pull in other dependencies while containing no code of their own.

To go into more detail about starters, let's pick this one: `spring-boot-starter-thymeleaf`. If we look at its `pom.xml` build file online (<https://github.com/spring-projects/spring-boot/tree/v1.1.6.RELEASE/spring-boot-starters/spring-boot-starter-thymeleaf/pom.xml>), we will see the following dependencies:

Dependency	What it provides
<code>spring-boot-starter</code>	A starter that brings in core dependencies that are critical for any Spring Boot-based project
<code>spring-boot-starter-web</code>	A starter that brings in embedded Tomcat, Jackson JSON binding, JSR 303 validation APIs, and Spring Web plus MVC support
<code>spring-core</code>	Critical parts of the Spring Framework (http://projects.spring.io/spring-framework)
<code>thymeleaf-spring4</code>	Core pieces of the Thymeleaf view engine along with Spring 4 integration
<code>thymeleaf-layout-dialect</code>	Thymeleaf dialect module



Wait, I thought this book was focused on Gradle! That's true, but Spring Boot itself is built with Maven. It is valuable to look at any of Spring Boot's starters in order to glean what they do.

Spring Boot is designed to help us build good apps rapidly. A key piece of making this happen is how Boot plugs in its opinion. When we include `spring-boot-starter-thymeleaf`, Spring Boot has the opinion that we'll probably want embedded Tomcat, Jackson JSON support, JSR 303 validation, and Spring Web MVC. So, it adds them as required dependencies.

Notice how there are no version numbers in the dependencies section? This is because each dependency we see listed is preset with a version number supplied by `spring-boot-gradle-plugin`. It's another opinion from Spring Boot about the best version of the library to use in conjunction with all the others.

The last thing Boot does is make auto-configuration decisions. The previous chapter showed us many examples of this, and it's happening here as well. Spring Boot configured view resolvers, an embedded servlet container, and other components that are commonly recommended for Spring MVC apps.

As we continue along this chapter, and throughout this book, we'll get to see more opinions that Boot inserts (and how it backs off when we make a different decision).



This amalgamation of libraries and chosen versions is known as **Spring IO** (<http://spring.io/platform>) and offers an out-of-the-box virtual collection of libraries that are verified to work together. Spring IO is very easy to use, as it's served up through the industry-standard Maven public repositories; it's not a downloadable bundle that becomes outdated the day after you get it.

Running a Spring Boot application

So far, we have a bare bones project. There isn't much code. However, `http://start.spring.io` creates a single Application class; let's look at that first:

```
package learningspringboot;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.context.annotation.ComponentScan;

@ComponentScan
@EnableAutoConfiguration
public class Application {

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

Let's break this down:

- `@ComponentScan`: This tells Spring to look for classes with `@Component`, `@Configuration`, `@Repository`, `@Service`, and `@Controller` and wire them into the app context as beans. By default, it scans for classes found underneath the package where the annotation is declared.
- `@EnableAutoConfiguration`: This turns on Boot's auto-configuration behavior.
- `public static void main()`: This uses Boot's `SpringApplication.run()` method as a convenient way to launch the app.

The `@EnableAutoConfiguration` key annotation is used for a Spring Boot-based application. It tells Boot to turn on all auto-configuration options. Each of these options looks at various aspects of the application and then makes decisions on adding extra beans. It makes decisions mostly based on the classpath and settings found inside `application.properties`.



Consider this example of auto-configuration. If Boot spots `JmsTemplate.class` on the classpath, it's an indication that the developer has added **spring-jms**. In this situation, Boot will automatically create an instance of `JmsTemplate` and make it available for injection to other Spring beans. The developer must still provide a `ConnectionFactory` bean. More details on how this works can be found in *Chapter 3, Debugging and Managing Your App*, where we will use `JmsAutoConfiguration` as an example.

Spring Boot's `SpringApplication.run()` method conveniently accepts a class as well as command-line arguments. In this case, it's plugging in the `Application` class, as this is the simplest way to build an app. We can start adding bean definitions right here or branch off in other places.

Adding Spring Social GitHub

We have discussed building an app that can scan GitHub repositories for open issues. The first step is to add a key project, which is **Spring Social GitHub**:

1. First, we need to visit `http://projects.spring.io/spring-social-github`.
2. From there, we can scroll down and find the latest release. In this case, we are using `1.0.0.BUILD-SNAPSHOT`, as M4 has some outstanding issues.
3. There's a slider that lets us pick our build system and shows us the content that we need to insert into our project's dependencies:

```
compile("org.springframework.social:spring-social-github:1.0.0.BUILD-SNAPSHOT")
```

4. This dependency will load Spring Social GitHub. Since it's not a general release, we need to add this to the `repositories` section:

```
repositories {
    mavenCentral()
    maven { url "https://repo.spring.io/libs-snapshot" }
}
```

By default, `start.spring.io` will include `mavenCentral` in the `repositories` section. In order to access Spring Social GitHub's `BUILD-SNAPSHOT` version, we had to add the second Maven URL.

With these two bits added to our build file, we are ready to build our app!

Digging into GitHub issues

Let's continue working on our simple app that will fetch GitHub issues from multiple repositories. To fetch GitHub issues, we need to establish a domain object:

```
package learningspringboot;

import org.springframework.social.github.api.GitHubIssue;

public class Issue {

    private String repo;
    private GitHubIssue githubIssue;

    public Issue(String repo, GitHubIssue githubIssue) {
        this.repo = repo;
        this.githubIssue = githubIssue;
    }

    public String getRepo() {
        return repo;
    }

    public GitHubIssue getGithubIssue() {
        return githubIssue;
    }
}
```

Spring Social GitHub comes with a `GitHubIssue` class, but this class doesn't include the name of the repository for a given issue. The `Issue` class listed in the preceding code is basically a wrapper that adds this extra bit of information. It's designed to be created through a constructor call that minimizes the risk of initializing it incompletely. It also includes getter calls in order to retrieve the data fields.

Next, we need a service that uses Spring Social GitHub's `GitHubTemplate` to retrieve issues:

```
package learningspringboot;

import java.util.ArrayList;
import java.util.List;

import org.springframework.social.github.api.GitHubIssue;
import org.springframework.social.github.api.impl.GitHubTemplate;
import org.springframework.stereotype.Service;

@Service
public class IssueManager {

    String githubToken =
        "ccdbf257f052a594a0e7bd2823a69ae38a48ffb1";

    String org = "spring-projects";

    String[] repos = new String[] { "spring-boot", "spring-boot-
        issues" };

    GitHubTemplate gitHubTemplate = new
        GitHubTemplate(githubToken);

    public List<Issue> findOpenIssues() {
        List<Issue> openIssues = new ArrayList<>();

        for (String repo : repos) {
            final List<GitHubIssue> issues = gitHubTemplate
                .repoOperations().getIssues(org, repo);

            for (GitHubIssue issue : issues) {
                if (issue.getState().equals("open")) {
                    openIssues.add(new Issue(repo, issue));
                }
            }
        }

        return openIssues;
    }
}
```


This class is marked as `@Service`, which means that it will be picked up and added to the app context by `@ComponentScan`. All of Spring's component annotations inherit from `@Component`, which gets them picked up by component scanning. It has a hardcoded GitHub passcode and a hardcoded organization name, and will fetch issues from `spring-boot` and `spring-boot-issues`. This service also has a `GitHubTemplate`.

The key function of this class, which is `findOpenIssues`, loops through the list of repositories, and then uses `GitHubTemplate` to retrieve open issues. It gathers them into a standard list, wrapped inside the `Issue` object we coded earlier.

Creating a GitHub access token

From where did we get this cryptic `githubToken` value of `ccdbf257f052a594a0e7bd2823a69ae38a48ffb1`? This is an OAuth access code that is required to plug in and talk to GitHub. To create one of your own, you need to create an account at <https://github.com>. After creating your account, perform the following steps:

1. Assuming the account is set up, visit <https://github.com/settings/applications>.
2. Scroll down until you see **Personal access tokens**.
3. Click on **Generate new token**. You'll probably be prompted to confirm your password.
4. Enter a description such as `Learning Spring Boot`, accept the default access controls, and click on **Generate token**.

You'll now see a newly minted cryptic token code with a copy-to-clipboard icon to the right. Grab it and paste it into your code, and you're ready! Have a look at the following screenshot:

Personal access tokens

Generate new token

Tokens you have generated that can be used to access the GitHub API.

Learning Spring Boot — <i>gist, repo, user</i>	Last used on June 16, 2014	Edit	Delete
iOctocat: Application — <i>delete_repo, gist, public_repo, user</i>	Last used on June 06, 2014	Edit	Delete
IntelliJ IDEA — <i>gist, repo, user</i>	No recent activity	Edit	Delete
hub — <i>repo</i>	Last used on May 05, 2014	Edit	Delete
guide-issues app — <i>repo</i>	Last used on June 16, 2014	Edit	Delete

Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to authenticate to the API over Basic Authentication.

If you thought all the hardcoded values weren't great, you're right. It is a bad design pattern. We'll remedy this design flaw later in this chapter.

The last bit of Java code that is required is a web controller that serves up a table of issues:

```
package learningspringboot;

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

@Controller
public class IssueController {

    private IssueManager issueManager;

    @Autowired
    public IssueController(IssueManager issueManager) {
        this.issueManager = issueManager;
    }

    @RequestMapping(value = "/")
    public String index(Model model) {
```

```
        model.addAttribute("issues",
            issueManager.findOpenIssues());
        return "index";
    }
}
```

The class is marked as a Spring MVC `@Controller`. The constructor is tagged `@Autowired`, so when the controller is created by the Spring container, it will initialize its `IssueManager` class using **constructor injection**.

The `index()` method is linked to the web route `/` through the `@RequestMapping` annotation. This particular endpoint includes a `Model` parameter, which is automatically supplied by Spring MVC. In this case, it invokes `issueManager.findOpenIssues()` and stores it in the model's `issues` entry. Then, it returns the name of the view to be rendered, which is `index.html`.



Constructor injection is currently the recommended way to wire Spring beans. It supports immutable bean configurations in a better manner and avoids beans getting partially configured at any particular time (<http://docs.spring.io/spring/docs/4.0.7.RELEASE/spring-framework-reference/htmlsingle/#beans-dependency-resolution>).

The last bit of code we need is a Thymeleaf template created at `src/main/resources/templates/index.html`:

```
<html xmlns:th="http://www.thymeleaf.org">
<body>
    <p>Open GitHub Issues</p>
    <table>
        <thead>
            <tr>
                <td>Repo</td>
                <td>Issue</td>
                <td>Title</td>
            </tr>
        </thead>
        <tbody>
            <tr th:each="issue : ${issues}">
                <td th:text="${issue.repo}"></td>
                <td>
                    <a th:href="${issue.githubIssue.url}"
                        target="_blank">
                        <span
                            th:text="${issue.githubIssue.number}"/>
                        </a>
                </td>
            </tr>
        </tbody>
    </table>
</body>
</html>
```

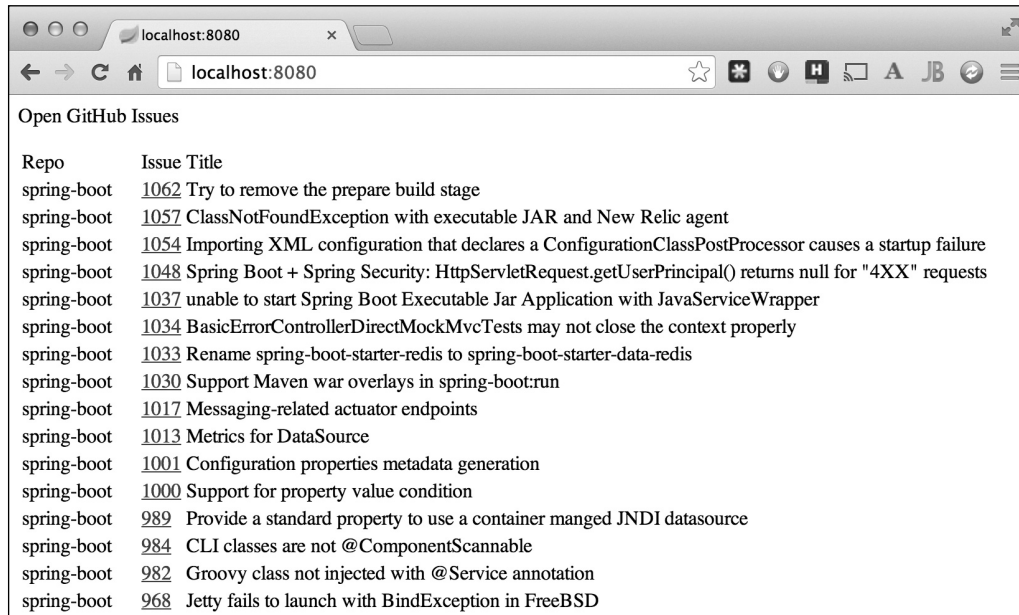


```
'beanNameVie...
2014-06-24 23:37:18.332 ... : JSR-330 'javax.inject.Inject' annotation
found a...
2014-06-24 23:37:18.750 ... : Server initialized with port: 8080
2014-06-24 23:37:18.993 ... : Starting service Tomcat
2014-06-24 23:37:18.993 ... : Starting Servlet Engine: Apache
Tomcat/7.0.54
2014-06-24 23:37:19.089 ... : Initializing Spring embedded
WebApplicationContext
2014-06-24 23:37:19.089 ... : Root WebApplicationContext: initialization
compl...
2014-06-24 23:37:19.578 ... : Mapping servlet: 'dispatcherServlet' to [/]
2014-06-24 23:37:19.581 ... : Mapping filter: 'hiddenHttpMethodFilter'
to: [//*]
2014-06-24 23:37:19.975 ... : Mapped URL path [/**/favicon.ico] onto
handler o...
2014-06-24 23:37:20.053 ... : Mapped "{[/],methods=[],params=[],headers=[
],con...
2014-06-24 23:37:20.056 ... : Mapped "{[/error],methods=[],params=[],head
ers=[...
2014-06-24 23:37:20.056 ... : Mapped "{[/error],methods=[],params=[],head
ers=[...
2014-06-24 23:37:20.082 ... : Mapped URL path [/**] onto handler of type
[clas...
2014-06-24 23:37:20.083 ... : Mapped URL path [/webjars/**] onto handler
of ty...
2014-06-24 23:37:20.464 ... : Registering beans for JMX exposure on
startup
2014-06-24 23:37:20.502 ... : Tomcat started on port(s): 8080/http
2014-06-24 23:37:20.504 ... : Started Application in 3.896 seconds (JVM
runnin...
```



What is **gradlew**? It's the **gradle wrapper**, which is a handy tool for any Gradle-based project. Let's assume that you have already downloaded Gradle from <http://www.gradle.org> or installed it using <http://gvmtool.net>. Inside your project, you can run `gradle wrapper`, and it will create a runnable environment with `gradlew` and `gradlew.bat` scripts. Push them out with your project, and your community won't be obligated to install Gradle. It also lets you control which version of Gradle is used for your project.

With the app running, we can visit `http://localhost:8080` and see the results.



Delving into Spring Boot's property support

In the previous section, we pointed out how hardcoding key attributes is a bad design pattern. Why? It makes it difficult to create updates. It also forces us to build different artifacts for every place our app might be deployed. Certain values such as `githubToken` that should remain secret become visible if our source code is released. Properties and environment variables provide a better alternative location to keep such sensitive and dynamic information.

Java properties and their associated files have existed for a long time, but Java's built-in APIs have always been clunky and require the developer to exert a lot of effort. This probably explains the relative lack of adoption by the industry. Spring Boot revitalizes the core idea behind properties, as we'll see.

The following code is an update to `IssueManager`:

```
package learningspringboot;

import java.util.ArrayList;
import java.util.List;
```

```
import org.springframework.beans.factory.InitializingBean;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.social.github.api.GitHubIssue;
import org.springframework.social.github.api.impl.GitHubTemplate;
import org.springframework.stereotype.Service;

@Service
public class IssueManager implements InitializingBean {

    @Value("${github.token}")
    String githubToken;

    @Value("${org}")
    String org;

    @Value("${repos}")
    String[] repos;

    GitHubTemplate gitHubTemplate;

    @Override
    public void afterPropertiesSet() throws Exception {
        this.gitHubTemplate = new GitHubTemplate(githubToken);
    }

    public List<Issue> findOpenIssues() {
        List<Issue> openIssues = new ArrayList<>();

        for (String repo : repos) {
            for (GitHubIssue issue : gitHubTemplate
                .repoOperations().getIssues(org, repo)) {
                if (issue.getState().equals("open")) {
                    openIssues.add(new Issue(repo, issue));
                }
            }
        }

        return openIssues;
    }
}
```

This version is almost the same as the previous `IssueManager` class. The difference is that we are using Spring's `@Value` annotation to glean properties instead of hardcoding them. Any of these newly defined properties can be injected from multiple sources. They are cascaded in the following order:

- Default values can be supplied directly with `@Value("${propertyName:defaultValue}")`
- `@Value` defaults can be overridden in an `application.properties` file, which gets bundled with the app in a JAR file
- Bundled properties can be overridden in an auxiliary `application.properties` file adjacent to the deployed JAR
- Auxiliary properties can be overridden by environment variables, either from the command line, a `.bashrc` file, or Windows environment settings
- In a cloud environment, environment variables can be supplied by the configuration, as we'll see toward the end of this chapter

Our code also creates `GitHubTemplate` using Spring's `InitializingBean` interface. The `afterPropertiesSet` method is called after all properties are configured by Spring's IoC container. This ensures that `githubToken` is populated when we create the template.

As we don't have default values, we need to create `src/main/resources/application.properties`:

```
org=spring-projects
repos=spring-boot,spring-boot-issues
```

Let's see what is happening:

- `org`: This is set to `spring-projects`. Our app's `GitHubTemplate` instance will use this to query `https://github.com/spring-projects`.
- `repos`: This is converted by Spring into `String[]{"spring-boot", "spring-boot-issues"}`.

As a bonus, Spring Boot provides relaxed rules on name binding. This means that we can set `githubToken` using either `github.token` or `GITHUB_TOKEN` as command-line environment variables. This provides universal support on *nix, Mac, and Windows. There is no need to write any code to process properties files!

Let's try out the latter approach:

```
$ GITHUB_TOKEN=ccdbf257f052a594a0e7bd2823a69ae38a48ffb1 ./gradlew clean bootRun
```


In firing up this app, we provided a command-line value for `GITHUB_TOKEN`. Our app is preloaded with an opinion on the settings. However, we have the flexibility to quickly override these values in production—should the need arise—and critical security details are kept out of our source code.



It's still the developer's responsibility to *not* add `github.token` to `src/main/resources/application.properties` and then push it to a publicly visible repository.

Adding server-side mobile support with Spring Mobile

So, we have a nicely functioning app that fetches GitHub issues. What can we do to take this example to the next level and explore Spring Boot? Considering that many companies are seeing their primary Internet traffic come from mobile consumers, what if we added mobile support to our app?

Spring Boot comes with out-of-the-box support for this. Just add the following dependency to `build.gradle`:

```
compile("org.springframework.boot:spring-boot-starter-mobile")
```

This pulls in **Spring Mobile** (<http://projects.spring.io/spring-mobile>), which is a library that easily switches between different views based on the browsing client's user agent.

First, add this line to `src/main/resources/application.properties`:

```
spring.mobile.devicedelegatingviewresolver.enabled=true
```

By default, mobile support is switched off. This value activates Spring Mobile's ability to switch views based on a user agent lookup.

Other settings are available, as listed in the following snippet:

```
spring.mobile.devicedelegatingviewresolver.normalPrefix=  
spring.mobile.devicedelegatingviewresolver.normalSuffix=
```

```
spring.mobile.devicedelegatingviewresolver.mobilePrefix=mobile/  
spring.mobile.devicedelegatingviewresolver.mobileSuffix=  
spring.mobile.devicedelegatingviewresolver.tabletPrefix=tablet/  
spring.mobile.devicedelegatingviewresolver.tabletSuffix=
```



These are the defaults inside Spring Boot and are not in our app.

These settings can be overridden either in our own `application.properties` file or later on, as described previously.



While the tablet prefix is configured, we are going to focus solely on mobiles in this chapter.

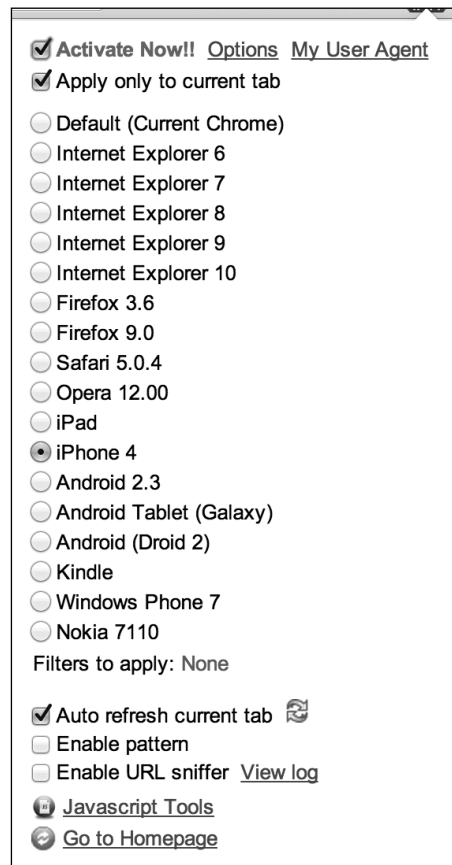
This is good. The default template, which is `src/main/resources/templates/index.html`, is the same as before. However, with the new `mobile/` prefix, we need to create `src/main/resources/templates/mobile/index.html`:

```
<html xmlns:th="http://www.thymeleaf.org">  
<body>  
    <p>Open GitHub Issues - Mobile</p>  
</body>  
</html>
```

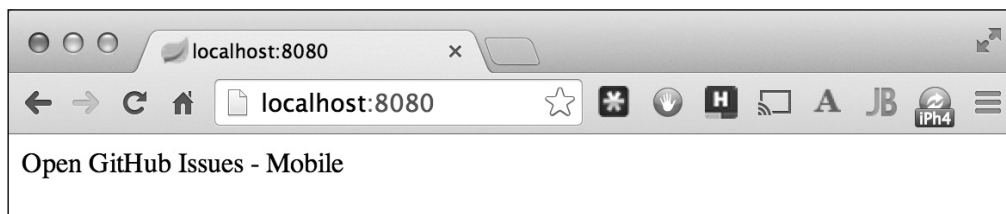
There's not much here. For the moment, it just displays an alternate message, which indicates that we hit the right view. We'll add to it later in this chapter. Let's just see how it switches views properly; it's time to fire it up.

Are we ready to test things out? Well, not quite yet. Spring Mobile uses the browser's user agent to make decisions based on the type of screen that is viewing the site. While we can use a real mobile device to test things out, this process can be quite cumbersome and time consuming. It's best to find a plugin for our browser in order to switch the user agent settings automatically.

Ultimate Agent Sniffer (<http://iblogbox.com/chrome/useragent/alert.php>) is one tool that lets you easily switch a browser tab to an iPad, iPhone, or just about any other device you can think of. However, it's not the only one. You can find one that suits your needs. The following screenshot shows you how to configure the current browser tab on an iPhone 4:



With the app running and our browser switched to mobile view, let's visit <http://localhost:8080>.



We can see the `mobile/index.html` template with ease. Things are lined up in order to create a truly mobile experience!

Creating a mobile frontend with jQuery Mobile

Mobile apps seem to be taking over the world. If a website doesn't provide a friendly mobile view, then people don't seem to like it. As we build web apps all the time, it's critical to create a frontend UI that is usable. jQuery Mobile is a handy toolkit that gets UIs up and running quickly.



This is merely a quick introduction to jQuery Mobile. For something more comprehensive, please read *jQuery Mobile Web Development Essentials* by Raymond Camden and Andy Matthews.

In the previous chapter, we took a quick glance at **Bower** (<http://bower.io>), which is a package manager for JavaScript libraries. We will use it to install jQuery Mobile into our app. Assuming that we have already installed Bower, let's proceed to define where Bower will put the packages by creating a `.bowerrc` file in the root of our project. This file will signal Bower to put all of our JavaScript modules into `src/main/resources/public/`, where Spring Boot can automatically serve them up:

```
{
  "directory": "src/main/resources/public/"
}
```

In the previous chapter, we dropped the files into `public/`. In this case, as we have a conventional Gradle project layout, the same target folder is at `src/main/resources/`. By the way, we have the same options (`/META-INF/resources/`, `/resources/`, `/static/`, `/public/`). We simply have to place them at `src/main/resources/`.

Looks like we're ready to go! Execute the following steps:

```
$ bower init
[?] name: issue-manager
[?] version: 0.1.0
[?] description: Learning Spring Boot - Issue Manager
[?] main file:
[?] what types of modules does this package expose? amd
```

```
[?] keywords:
[?] authors: Greg Turnquist <gturnquist@pivotal.io>
[?] license: ASL
[?] homepage: http://blog.gregturnquist.com/category/learning-spring-
boot
[?] set currently installed components as dependencies? No
[?] add commonly ignored files to ignore list? Yes
[?] would you like to mark this package as private, which prevents it
from being accidentally published to the registry? Yes
...
[?] Looks good? Yes
```


With this setup, let's now install jQuery Mobile:

```
$ bower install jquery-mobile-bower --save
bower jquery-mobile-bower#*      cached git://github.com/jobrapido/jquery-
mobile-bower.git#1.4.2
bower jquery-mobile-bower#*      validate 1.4.2 against git://github.com/
jobrapido/jquery-mobile-bower.git#*
bower jquery#~1.10.0             cached git://github.com/jquery/jquery.
git#1.10.2
bower jquery#~1.10.0             validate 1.10.2 against git://github.com/
jquery/jquery.git#~1.10.0
bower jquery-mobile-bower#~1.4.2      install jquery-mobile-
bower#1.4.2
bower jquery#~1.10.0             install jquery#1.10.2

jquery-mobile-bower#1.4.2 src/main/resources/public/jquery-mobile-bower
└─ jquery#1.10.2

jquery#1.10.2 src/main/resources/public/jquery
```

We can see that it installed jQuery Mobile 1.4.2 along with jQuery 1.10.0. This gives us all we need in order to craft a mobile web page.

 Again, this isn't a complete introduction to jQuery Mobile. A great resource for learning how to drive various widgets is jQuery Mobile's showcase, which is available at <http://demos.jquerymobile.com/1.4.2>.

To see our complete lineup of JavaScript modules for the frontend, check out `bower.json`:

```
{
  "name": "issue-manager",
  "version": "0.1.0",
  "authors": [
    "Greg Turnquist <gtturnquist@pivotal.io>"
  ],
  "description": "Learning Spring Boot - Issue Manager",
  "license": "ASL",
  "homepage": "http://blog.gregturnquist.com/category/learning-spring-boot",
  "private": true,
  "ignore": [
    "**/*.*",
    "node_modules",
    "bower_components",
    "src/main/resources/public/",
    "test",
    "tests"
  ],
  "dependencies": {
    "jquery-mobile-bower": "~1.4.2"
  }
}
```

It's time to put the pedal to the metal and load up jQuery Mobile's CSS and JavaScript components. The following page is a very simple mobile layout that we need to create at `src/main/resources/templates/mobile/index.html`:

```
<html xmlns:th="http://www.thymeleaf.org">
<head>
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <link rel="stylesheet" href="jquery-mobile-bower/css/jquery.mobile-1.4.2.css" />
  <script src="jquery/jquery.js"></script>
  <script src="jquery-mobile-bower/js/jquery.mobile-1.4.2.js"></script>
</head>
<body>
  <div data-role="page" id="home">

    <div data-role="header" data-position="fixed">
```

```
        <a href="/" data-icon="home" data-
            iconpos="notext"></a>
        <h1>Issue Manager Mobile</h1>
    </div>

    <div data-role="content">
        <ul data-role="listview">
            <li th:each="issue : ${issues}">
                <a th:href="${issue.githubIssue.htmlUrl}"
                    target="_blank"
                    th:text="${issue.githubIssue.title}">
                </a>
            </li>
        </ul>
    </div>

</div>
</body>
</html>
```

Where do we begin? Okay, maybe it's a little bit bigger than you expected. However, if you've done any major hacking on HTML, you might recognize that this isn't as huge as other frontend systems. So, let's break it up:

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

The first line is a giveaway that this is most certainly a Thymeleaf template. It declares the `th` namespace, and we plan to take full advantage of this later in the code:

```
<head>
    <meta name="viewport" content="width=device-width, initial-
        scale=1" />
    <link rel="stylesheet" href="jquery-mobile-
        bower/css/jquery.mobile-1.4.2.css" />
    <script src="jquery/jquery.js"></script>
    <script src="jquery-mobile-bower/js/jquery.mobile-
        1.4.2.js"></script>
</head>
```

The header section starts off by creating some viewport settings. Essentially, it's saying that the browser should use the full width of the device and that we are completely zoomed in. Double-tapping the screen of your phone won't cause it to zoom in any more. Next, we are loading up jQuery Mobile's CSS style sheet. Then, we load up jQuery and jQuery Mobile JavaScript modules.

Now, let's look at the `<body>` part of the page:


```
<body>
  <div data-role="page" id="home">

    <div data-role="header" data-position="fixed">
      <a href="/" data-icon="home" data-
        iconpos="notext"></a>
      <h1>Issue Manager Mobile</h1>
    </div>

    <div data-role="content">
      <ul data-role="listview">
        <li th:each="issue : ${issues}">
          <a th:href="${issue.githubIssue.htmlUrl}"
            target="_blank"
            th:text="${issue.githubIssue.title}">
          </a>
        </li>
      </ul>
    </div>

  </div>
</body>
</html>
```

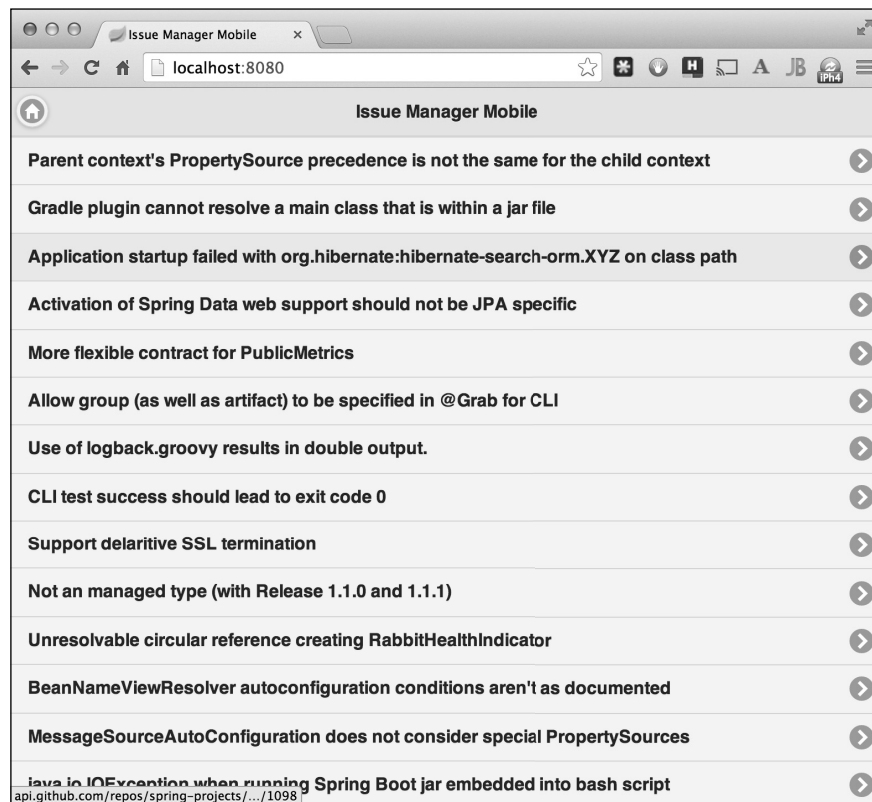
jQuery Mobile is a declarative toolkit, which means that we only have to lay out a set of key elements, and when the package finishes loading it will apply mobile CSS styling. In this case, we have a top level `<div data-role="page" id="home">` tag, which defines a mobile "page." The term "page" is wrapped in scary quotes because it isn't the same thing as an HTML page. jQuery Mobile can support multiple pages and will only display the current one.

 As this is a single page app, we aren't going any deeper into multiple pages. Instead, we'll focus on the other parts.

Inside the "home" page, there is a `<div data-role="header" data-position="fixed">` tag. This fragment defines what appears at the top. Basically, a header can show up to three components on the device: the left, the middle, and the right. In our case, there is an anchor tag and a header. This automatically gets shifted to the left and middle spots upon rendering. The anchor tag will be rendered as a button but with a home icon instead of any text. The header tag takes the middle slot. For our example, there isn't anything that can be put in the right slot.

After this, we get to the meat of the page: `<div data-role="content">`. This is the content and is where most of the device's real estate will be put to work. It's very analogous to the desktop version of things. The exception is that instead of a table with rows, we are creating a list view of listed items. jQuery Mobile will convert every line item into a button.

The anchor tag inside the list item has the URL. When you click on one of these mobile buttons, it will open a new tab in our browser. The button's text shows you the title as the text value.



This seems to do the trick! Using our desktop browser and Ultimate User Agent Switcher has made it easy to build this mobile frontend. However, nothing is complete without a check from a *real* mobile device. We'll see how to do this a bit later.

Bundling up the application as a runnable JAR

In the previous chapter, we learned how to run Groovy scripts with Spring Boot's CLI tool. This empowered us to create runnable JAR files, which can be deployed anywhere a JVM is installed. In this chapter, let's see how a Java-based project can be bundled up as a JAR and deployed to a popular PaaS provider.

Spring Boot comes with two handy plugins: `spring-boot-maven-plugin` and `spring-boot-gradle-plugin`. As we are using Gradle in this book, the project file from `start.spring.io` has `spring-boot-gradle-plugin` installed. Earlier, we ran the app from the command line using `./gradlew bootRun`. To bundle up a JAR file, we merely need to do this:

```
$ ./gradlew clean build
:clean
:compileJava
:processResources
:classes
:jar
:bootRepackage
:assemble
:compileTestJava UP-TO-DATE
:processTestResources UP-TO-DATE
:testClasses UP-TO-DATE
:test UP-TO-DATE
:check UP-TO-DATE
:build

BUILD SUCCESSFUL

Total time: 7.168 secs
```



Did you get an error from Gradle? `start.spring.io` creates `ApplicationTests`, which will fail due to the required `github.token` property. As we aren't focused on writing automated tests, the simplest thing to do is to delete this test class and try again. Pay heed to the fact that skipping unit tests is *not* recommended for a real production app.

Spring Boot initially builds a traditional JAR file. This file contains the compiled class files, all the public resource files such as our jQuery Mobile code and HTML templates, and the pom file. We can find it at `build/libs/issue-manager-0.0.1-SNAPSHOT.jar.original`. This JAR file isn't runnable. In fact, it doesn't even have third-party dependencies; this is by design. Such a JAR file can only be used to build a bigger artifact, such as a WAR file.

In the spirit of runnable apps, Spring Boot's plugin takes another step. It creates a new JAR file based on the original one and then adds third-party dependencies and some support code in order to load the libraries. This can be found at `build/libs/issue-manager-0.0.1-SNAPSHOT.jar`:

```
$ ls build/libs
issue-manager-0.0.1-SNAPSHOT.jar
issue-manager-0.0.1-SNAPSHOT.jar.original
```

We can fire up the JAR file now:

```
$ GITHUB_TOKEN=ccdbf257f052a594a0e7bd2823a69ae38a48ffb1 java -jar build/
libs/i...
```

```

      _ _ _ _ _
  /\ /  _ ' _ _ _ ( ) _ _ _ _ \ \ \ \
 ( ( ) \ _ | ' _ | ' _ | ' _ \ / _ | \ \ \ \
 \ \ / _ ) | | _ | | | | | | ( | | ) ) )
   ' | _ | . _ | | _ | | \ , | / / / /
=====|_|=====|_|/_/_/_/_/
:: Spring Boot ::          (v1.1.6.RELEASE)
```

```
2014-06-24 23:48:38.119 ... : Starting Application on retina with PID
10542 (/...
2014-06-24 23:48:38.182 ... : Refreshing org.springframework.boot.
context.embe...
2014-06-24 23:48:38.775 ... : Overriding bean definition for bean
'beanNameVie...
```

```

2014-06-24 23:48:39.278 ... : JSR-330 'javax.inject.Inject' annotation
found a...
2014-06-24 23:48:39.864 ... : Server initialized with port: 8080
2014-06-24 23:48:40.156 ... : Starting service Tomcat
2014-06-24 23:48:40.157 ... : Starting Servlet Engine: Apache
Tomcat/7.0.54
2014-06-24 23:48:40.288 ... : Initializing Spring embedded
WebApplicationContext
2014-06-24 23:48:40.289 ... : Root WebApplicationContext: initialization
compl...
2014-06-24 23:48:40.935 ... : Mapping servlet: 'dispatcherServlet' to [/]
2014-06-24 23:48:40.938 ... : Mapping filter: 'hiddenHttpMethodFilter'
to: [//*]
2014-06-24 23:48:41.687 ... : Mapped URL path [/**/favicon.ico] onto
handler o...
2014-06-24 23:48:41.803 ... : Mapped "{[/],methods=[],params=[],headers=[
],con...
2014-06-24 23:48:41.805 ... : Mapped "{[/error],methods=[],params=[],head
ers=[...
2014-06-24 23:48:41.805 ... : Mapped "{[/error],methods=[],params=[],head
ers=[...
2014-06-24 23:48:41.831 ... : Mapped URL path [/**] onto handler of type
[clas...
2014-06-24 23:48:41.831 ... : Mapped URL path [/webjars/**] onto handler
of ty...
2014-06-24 23:48:42.042 ... : Registering beans for JMX exposure on
startup
2014-06-24 23:48:42.106 ... : Tomcat started on port(s): 8080/http
2014-06-24 23:48:42.107 ... : Started Application in 4.513 seconds (JVM
runnin...

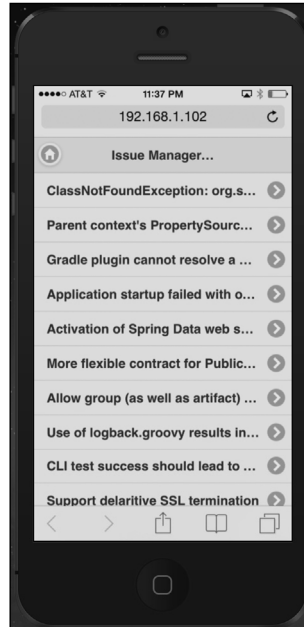
```

Now we can visit <http://localhost:8080> and either view the desktop version or the mobile version. If we check for the local IP address of our machine, we can view it from a mobile device that's on the same Wi-Fi network.



If you are on a Mac or *nix system, it's possible to find the local IP address by typing `ifconfig | grep inet | grep -v 127.0.0.1`.

Assuming that we get our computer and phone on the same network, we can easily view the mobile version of the app.



We can see the list of tickets open on the GitHub repositories. If we click on one, it will take us to GitHub, where we can easily view the details of the issue.

Deploying to Cloud Foundry

One popular PaaS (Platform as a Service) provider is Cloud Foundry. You can visit a public-facing version known as Pivotal Web Services at <https://run.pivotal.io> to discover options. It's also possible for you to build and set up your own local instance of Cloud Foundry using the open source tools found at <https://github.com/cloudfoundry>, but we won't go into that. Assuming that you have an account at run.pivotal.io, let's proceed by installing the Cloud Foundry CLI tool.

If we visit <https://github.com/cloudfoundry/cli>, we can find installation instructions for multiple platforms.

On a Mac system with Homebrew (<http://brew.sh>), all we have to do is type the following:

```
$ brew tap pivotal/tap
$ brew install cloudfoundry-cli
```

```
==> Downloading https://downloads.sf.net/project/machomebrew/Bottles/
cloudfoundry-cli-6.1.1.mavericks.bottle.tar.gz
```

```
Already downloaded: /Library/Caches/Homebrew/cloudfoundry-cli-
6.1.1.mavericks.bottle.tar.gz
```

```
==> Pouring cloudfoundry-cli-6.1.1.mavericks.bottle.tar.gz
```

From here, we can log in to CF:

```
$ cf login
```

```
API endpoint: https://api.run.pivotal.io
```

```
Email> gturnquist@pivotal.io
```

```
Password>
```

```
Authenticating...
```

```
OK
```

```
API endpoint: https://api.run.pivotal.io (API version: 2.6.0)
```

```
User:          gturnquist@pivotal.io
```

```
Org:           FrameworksAndRuntimes
```

```
Space:         development
```

We are prompted for the API endpoint. In Pivotal's commercial instance of CF, this will be `https://api.run.pivotal.io`. If you are running a different instance, your API endpoint will be different. We must then use our e-mail/password credentials. After getting in, we might have to pick our organization and space for deployment. In all likelihood, your options will be different than what's shown in the preceding console output.

At this point, we can deploy our app as follows:

```
$ cf push issue-manager-gturnquist -p target/issue-manager-0.0.1-
SNAPSHOT.jar -m 512M
```

```
Creating app issue-manager-gturnquist in org FrameworksAndRuntimes /
space development as gturnquist@pivotal.io...
```

```
OK
```

```
Creating route issue-manager-gturnquist.cfapps.io...
```

```
OK
```

```
Binding issue-manager-gturnquist.cfapps.io to issue-manager-gturnquist...
```

OK

Uploading issue-manager-gturnquist...

Uploading app files from: target/issue-manager-0.0.1-SNAPSHOT.jar

Uploading 1.7M, 474 files

OK

Starting app issue-manager-gturnquist in org FrameworksAndRuntimes /
space development as gturnquist@pivotal.io...

OK

-----> Downloaded app package (13M)

-----> Java Buildpack Version: v2.1.2 | <https://github.com/cloudfoundry/java-buildpack.git#074fd9a>

-----> Downloading Open Jdk JRE 1.8.0_60 from http://download.run.pivotal.io/openjdk/lucid/x86_64/openjdk-1.8.0_60.tar.gz (1.4s)

Expanding Open Jdk JRE to .java-buildpack/open_jdk_jre (1.0s)

-----> Downloading Spring Auto Reconfiguration 0.8.9 from <http://download.run.pivotal.io/auto-reconfiguration/auto-reconfiguration-0.8.9.jar> (0.0s)

-----> Uploading droplet (44M)

0 of 1 instances running, 1 starting

...

0 of 1 instances running, 1 down

0 of 1 instances running, 1 failing

FAILED

Start unsuccessful

NOTE: use 'cf logs issue-manager-gturnquist --recent' for more
information

First, let's examine the arguments used to push our app to CF:

- The app name is issue-manager-gturnquist.
- The artifact was supplied with -p target/issue-manager-0.0.1-SNAPSHOT.jar.
- The memory was increased to 512 MB. It has been observed that Spring Boot apps need more than the minimum memory for web-based apps. (Why? I'm not sure at the time of writing this.)

However, wait a second; `cf` says that the deployment failed! Why is that? Well, it appears to be giving us a clue by showing us how to check the logfiles:

```
$ cf logs issue-manager-gturnquist --recent
...
2014-06-15T00:35:23.45-0500 [App/0]    ... Could not resolve placeholder
'github.token' in string value "${github.token}"
...
```

Ah ha! We don't have the `github.token` property configured. In all the other examples of running this app, whether using `./gradlew bootRun` or by running the executable JAR file, we supplied that value. So what do we do in this situation? Simple:

```
$ cf set-env issue-manager-gturnquist GITHUB_TOKEN
ccdbf257f052a594a0e7bd2823a69ae38a48ffb1
Setting env variable 'GITHUB_TOKEN' to
'ccdbf257f052a594a0e7bd2823a69ae38a48ffb1' for app issue-manager-
gturnquist...
```

OK

NOTE: Use `'cf push'` to ensure your env variable changes take effect

Okay, we've supplied an environmental variable that will be associated with the cloud-hosted instance of this app. We can push updated JAR files all we want. As long as we don't delete the remote app flat out, its environmental settings will stick:

```
$ cf push issue-manager-gturnquist -p target/issue-manager-0.0.1-
SNAPSHOT.jar
```

```
...
0 of 1 instances running, 1 starting
0 of 1 instances running, 1 starting
0 of 1 instances running, 1 starting
1 of 1 instances running
```

App started

```
Showing health and status for app issue-manager-gturnquist in org
FrameworksAndRuntimes / space development as gturnquist@pivotal.io...
```

OK

```
requested state: started
```

```
instances: 1/1
```


usage: 512M x 1 instances

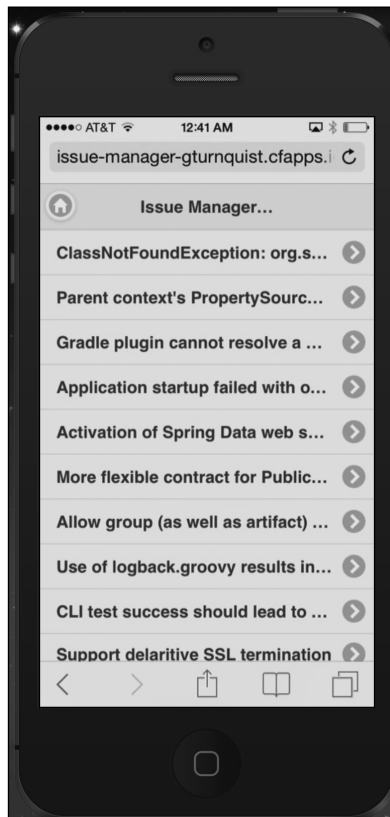
urls: issue-manager-gturnquist.cfapps.io

	state	since	cpu	memory	disk
#0	running	2014-06-15 12:38:26 AM	0.0%	251.4M of 512M	108.3M of 1G

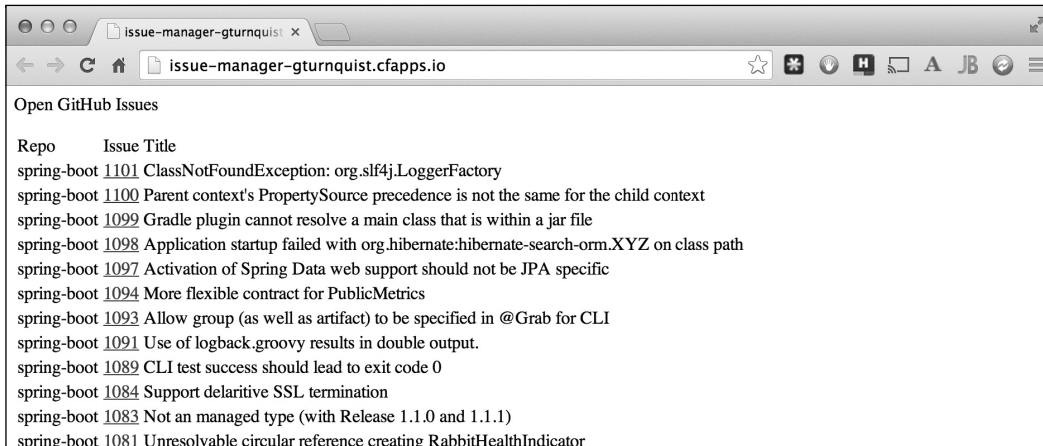


We didn't supply the memory argument this time, because that information is already stored with the app, just like `GITHUB_TOKEN`.

Now, we can visit <http://issue-manager-gturnquist.cfapps.io>.



Perfect! This looks identical to what we saw earlier, except that we don't have to visit port 8080, and it's available anywhere on the Internet.



If we visit the same site from our desktop browser, we can see the original look and feel.

Adding production-ready support

So, is anything missing? Well, in the previous chapter, we added `@Grab("spring-boot-actuator")` and `@Grab("spring-boot-starter-remote-shell")`, and it created a slew of extra HTTP endpoints. While Gradle isn't quite as concise as Groovy Grape, it's not that hard. Just add the following dependencies to `build.gradle`:

```
compile("org.springframework.boot:spring-boot-starter-actuator")
compile("org.springframework.boot:spring-boot-starter-remote-shell")
```

These two Spring Boot starters activate Spring Boot Actuator as well the CRaSH remote shell support.

In the previous chapter, we saw some detailed screenshots and explored how to view these endpoints in our browser. All this is quite useful. However, one thing that was mentioned was the possibility of writing a script in order to consume the metrics. What would it take to start gathering metric data every second and dump it into a CSV file that can be read with Excel? Let's start by creating an independent script called `metrics.groovy` in the root folder of our project:

```
package learningspringboot

@Grab("groovy-all")
import groovy.json.*

@EnableScheduling
class MetricsCollector {

    def url = "http://localhost:8080/metrics"
    def slurper = new JsonSlurper()
    def keys = slurper.parse(new URL(url)).keySet()
        .findAll{
            it.startsWith("counter")
        }
    def header = false;

    @Scheduled(fixedRate = 1000L)
    void run() {
        if (!header) {
            println(keys.join(','))
            header = true
        }

        def metrics = slurper.parse(new URL(url))

        println(keys.collect{metrics[it]}.join(','))
    }
}
```

So, what is buried in this gem of a script? Take a look:

- `@Grab("groovy-all")` brings the vast wealth of Groovy on board (most notably, `JsonSlurper`)
- We initialize a connection to `http://localhost:8080/metrics` and then fetch all keys that start with `counter`

- `@EnableScheduling` turns on Spring's ability to have scheduled method calls; `run()` is booked to run every 1000 ms
- The first time `run()` executes, it will print out a header with each key's name joined by commas (the CSV format)
- Every other time `run()` is executed, it reconnects to `http://localhost:8080/metrics` in order to download and parse the data
- Finally, using the list of keys, each data point is gathered and printed out, spliced together by commas (CSV)

To use the script, we first need to fire up our new and improved Issue Manager app with Spring Boot Actuator turned on:

```
$ GITHUB_TOKEN=ccdbf257f052a594a0e7bd2823a69ae38a48ffb1 ./gradlew clean bootRun
```

With the Actuator up and running, we can now launch our metrics collection script in another shell (in the same folder in which `metrics.groovy` lives):

```
$ spring run -q metrics.groovy | tee metrics.csv
counter.status.200.autoconfig,counter.status.200.beans,counter.status.200.conf...
1,1,1,1,1,1,1,1,1,21,1,2,1
1,1,1,1,1,1,1,1,1,22,1,2,1
1,1,1,1,1,1,1,1,1,23,1,2,1
1,1,1,1,1,1,1,1,1,24,1,2,1
1,1,1,1,1,1,1,1,1,25,1,2,1
1,1,1,1,1,1,1,1,1,26,1,2,1
1,1,1,1,1,1,1,1,1,27,1,2,1
1,1,1,1,1,1,1,1,1,28,1,2,1
1,1,1,1,1,1,1,1,1,29,1,2,1
1,1,1,1,1,1,1,1,1,30,1,2,1
1,1,1,1,1,1,1,1,1,31,1,2,1
1,1,1,1,1,1,1,1,1,32,1,2,1
1,1,1,1,1,1,1,1,1,33,1,2,1
1,1,1,1,1,1,1,1,1,34,1,2,1
1,1,1,1,1,1,1,1,1,35,1,2,1
1,1,1,1,1,1,1,1,1,36,1,2,1
```

Assuming we have done that, we can now open `metrics.csv` using Excel.

	A	B	C	D	E	F	G	H	I	J	K	L
1	counter.stat	counter.stat	counter.stat	counter.stat	counter.stat	counter.stat	counter.stat	counter.stat	counter.stat	counter.stat	counter.stat	counter.status.200.trace
2	1	1	1	2	1	1	1	1	1	59	3	1
3	1	1	1	2	1	1	1	1	1	60	3	1
4	1	1	1	2	1	1	1	1	1	61	3	1
5	1	1	1	2	1	1	1	1	1	62	3	1
6	1	1	1	2	1	1	1	1	1	63	3	1
7	1	1	1	2	1	1	1	1	1	64	3	1
8	1	1	1	2	1	1	1	1	1	65	3	1
9	1	1	1	2	1	1	1	1	1	66	3	1
10	1	1	1	2	1	1	1	1	1	67	3	1
11	1	1	1	2	1	1	1	1	1	68	3	1
12	1	1	1	2	1	1	1	1	1	69	3	1
13	1	1	1	2	1	1	1	1	1	70	3	1
14	1	1	1	2	1	1	1	1	1	71	3	1
15	1	1	1	2	1	1	1	1	1	72	3	1
16	1	1	1	2	1	1	1	1	1	73	3	1
17												
18												
19												

Gee, that's no fun. The only metrics that are increasing appear to be our script hitting `/metrics`. Well, duh! If only there was a way to automatically visit other sites. Oh, but there is! Groovy is a powerful platform. Why don't we write a script that can load test our website at the same time? Let's do this as follows:

```
package learningspringboot

@Grab('org.codehaus.gpars:gpars:1.1.0')
import groovyx.gpars.GParsPool
import groovy.util.logging.*

@Slf4j
class LoadTester implements CommandLineRunner {

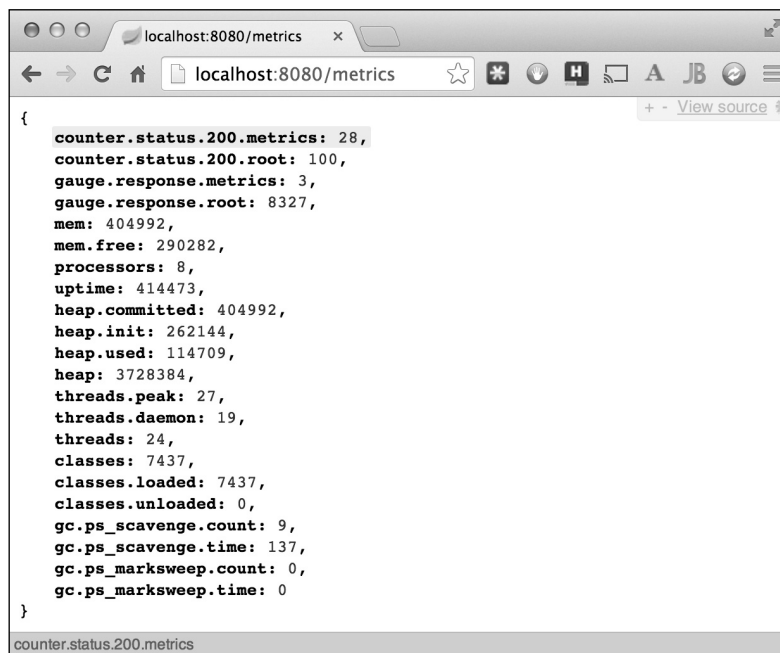
    void run(String[] args) {
        GParsPool.withPool(8) {
            def loadset = ["http://localhost:8080"]*100
            loadset.eachParallel { url ->
                def results = url.toURL().text
                log.info("Hit ${url}")
            }
        }
    }
}
```

So, what's all this? It's not that long, but it's packed with lots of features:

- GPar (http://gpars.codehaus.org/) is the Groovy Parallel Systems library. We're using it to perform parallel load testing
- This script implements Spring Boot's `CommandLineRunner` interface, so it will invoke the `run()` method
- `GParPool.withPool(8)` creates a parallel pool with eight workers (because I happen to have an eight-core laptop)
- Inside this block, we create an array with 100 instances of `http://localhost:8080`, which is the root URL of issue manager
- With this array of addresses, we ask GPar to iterate over every entry using `eachParallel { }` in parallel
- Inside each iteration, we convert `url` into `java.net.URL`, connect, and fetch the text of the page
- Instead of printing out the content, we are effectively just clicking on the URL

This effectively hits the website 100 times, as fast as eight cores can go. Looking at the following metrics screenshot, we can see that `counter.status.200.root` is now up to 100.

With this setup, let's launch `spring run metrics.groovy | tee metrics.csv` in one shell while we run `spring run load_test.groovy` in another.



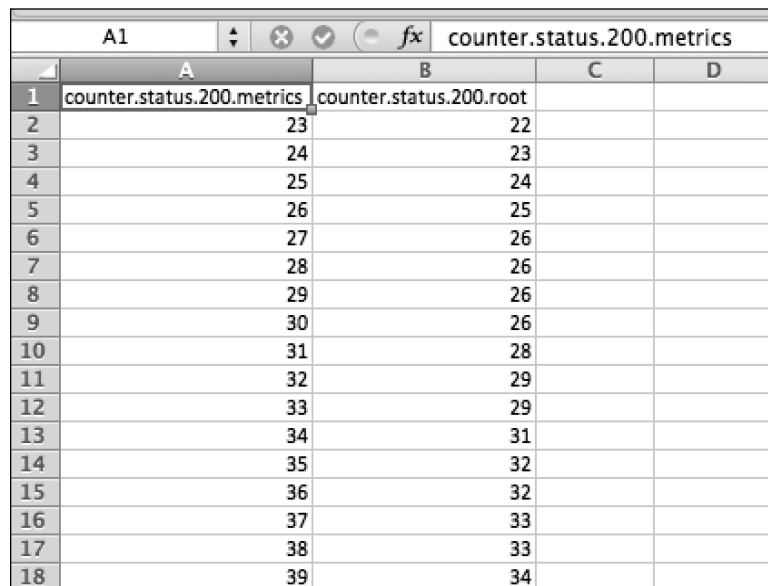
```

{
  counter.status.200.metrics: 28,
  counter.status.200.root: 100,
  gauge.response.metrics: 3,
  gauge.response.root: 8327,
  mem: 404992,
  mem.free: 290282,
  processors: 8,
  uptime: 414473,
  heap.committed: 404992,
  heap.init: 262144,
  heap.used: 114709,
  heap: 3728384,
  threads.peak: 27,
  threads.daemon: 19,
  threads: 24,
  classes: 7437,
  classes.loaded: 7437,
  classes.unloaded: 0,
  gc.ps_scavenge.count: 9,
  gc.ps_scavenge.time: 137,
  gc.ps_marksweep.count: 0,
  gc.ps_marksweep.time: 0
}

```

counter.status.200.metrics

This screenshot nicely shows our metrics counter results, gathered in a persistent spreadsheet:



	A1				fx	counter.status.200.metrics
	A	B	C	D		
1	counter.status.200.metrics	counter.status.200.root				
2	23	22				
3	24	23				
4	25	24				
5	26	25				
6	27	26				
7	28	26				
8	29	26				
9	30	26				
10	31	28				
11	32	29				
12	33	29				
13	34	31				
14	35	32				
15	36	32				
16	37	33				
17	38	33				
18	39	34				

Here, we can see that `metrics.csv` has counts of both `counters.status.200.metrics` and `counters.status.200.root`.

For production-ready support, it's nice that Actuator provides us handy data that we can consume. The scripts that we used in order to consume these metrics might seem a bit crude. However, in the real world, we need tools that let us quickly try something out before investing in bigger, more complex, and often more expensive solutions.

If our manager wanted a quick read of the last 24 hours of data, we could easily take `metrics.groovy` and adjust it to have a rolling time limit. We could then serve up the content using a little Spring MVC. Cash in on Spring MVC's WebSocket support, and we could have the screens update dynamically. Program managers love eye candy, right?

There's no telling what requirements we'll get from system administrators and managers. Spring Boot's easy-to-consume endpoints provide us with the tools we need in our main application as well as the support tools we'll build in order to manage things.

Summary

In this chapter, we used `http://start.spring.io` to create a bare bones project with Gradle support. We plugged in Spring Social GitHub and used it to scan multiple repositories for open issues. Then, we used Spring Mobile and jQuery Mobile to create an alternative mobile frontend. We learned how to use Spring Boot's über easy property support. We bundled it up into a runnable JAR file and deployed it to Cloud Foundry. Finally, we added production-ready support and did a little bit of load testing while gathering metrics.

In the next chapter, we will explore all the tools at our disposal that can be used to debug and manage Spring Boot applications.

Where to buy this book

You can buy Learning Spring Boot from the Packt Publishing website:
<https://www.packtpub.com/application-development/learning-spring-boot>.

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our [shipping policy](#).

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.



www.PacktPub.com

For More Information:

www.packtpub.com/application-development/learning-spring-boot