



```
// Create the Application  
window.IssueTrackerApp = new
```

```
// Create the top-level Region  
IssueTrackerApp.addRegions({  
  headerRegion : '#header-reg  
  mainRegion   : '#main-regio  
  dialogRegion : '#dialog-reg  
  footerRegion : '#footer-reg  
});
```

```
// Application start Callback  
IssueTrackerApp.on('start', function(options) {  
  // Initialize the Router  
  logger.debug("Backbone.history.start");  
  Backbone.history.start();  
});
```

```
// Launch the Issue List  
if(IssueTrackerApp.getCurrentRoute() === '/') {  
  IssueTrackerApp.execute('issuemanager:list');  
}  
});
```

 HTML5 JavaScript Mobile Web Services Database Full Stack

```
ion();
```

```
@RequestMapping(  
    value = "/issues",  
    method = RequestMethod.GET,  
    produces = MediaType.APPLICATION_JSON_VALUE)  
public ResponseEntity<List<Issue>> getAllIssues() {  
    logger.info("> getAllIssues");  
  
    List<Issue> issues = null;  
    try {  
        issues = issueService.findAll();  
  
        // if no issues found, create an empty list  
        if (issues == null) {  
            issues = new ArrayList<Issue>();  
        }  
    } catch (Exception e) {  
        logger.error("Unexpected Exception caught.", e);  
        return new ResponseEntity<List<Issue>>(issues,  
            HttpStatus.INTERNAL_SERVER_ERROR);  
    }  
  
    logger.info("< getAllIssues");  
    return new ResponseEntity<List<Issue>>(issues, HttpStatus.OK);  
}
```

Lean Application Engineering featuring Backbone.Marionette.js and the Spring Framework



Learn lean application engineering techniques building a single page web application and RESTful web services using Backbone.Marionette.js and the Spring Framework.

Lean Application Engineering

Featuring Backbone.Marionette and the Spring Framework

Matthew Warman

This book is for sale at <http://leanpub.com/leanstacks-marionette-spring>

This version was published on 2015-06-29



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

©2014 - 2015 Matthew Warman

Tweet This Book!

Please help Matthew Warman by spreading the word about this book on [Twitter](#)!

The suggested hashtag for this book is [#leanstacks](#).

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

<https://twitter.com/search?q=#leanstacks>

Contents

Part I. Introduction	i
Paradigm Shift	ii
About the Book	iv
Intended Audience	v
Git Repositories	vi
Web Services	vi
Single Page Application	vi
 Part II. Server Foundation	 1
Bootstrap the Server Project	2
A Proper Foundation	2
Installing the Dependencies	2
Choosing a Source Editor	4
Structuring the Project	6
Getting Started with Spring Boot	15
Updating the POM	15
Application.java	17
Starting the Server	18
Building the First RESTful Web Service	22
Cohesion	22
Updating the POM	22
Creating the Entity Model	23
Creating the Repository	29
Creating the Service	30
Creating the Controller	33
Creating the Cross-Origin Filter	37
Running the Application	39

CONTENTS

Part III. User Interface Foundation	42
Bootstrap the User Interface Project	43
Installing the Dependencies	43
Choosing a Source Editor	45
Structuring the Project	45
Getting Started with Marionette	57
The Marionette Application	57
Marionette Modules	62
Displaying Static Content	63
Organizing the Application	75
Marionette Messaging	77
Event Aggregator	77
Commands	78
Request Response	79
Integrating with Web Services	81
Creating the Issue Entity	81
Creating the Issue Manager Module	85
Listing Issues at Startup	96
Running the Application	97
Part IV. CRUD	100
Creating Issues	101
Web Service	101
User Interface	103
Running the Application	117
Viewing Issues	120
Web Services	120
User Interface	122
Running the Application	135
Updating Issues	138
Web Service	138
User Interface	140
Running the Application	154
Deleting Issues	157
Web Services	157
User Interface	159

CONTENTS

Running the Application	165
Part V. Production Features	168
Development Operations	169
Web Services	169
User Interface	174
Deployment Strategies	179
Configuration	183
Property Files	183
Property File Load Order	185
Using Profiles with Property Files	186
Spring Boot Property Reference	187
Production Databases	189
Relational Databases	189
NoSQL Databases	194
Production Monitoring and Management	206
Introducing Actuator	206
Enabling Actuator	206
Endpoints	207
Configuration	208
Health Check	209
Application Information	211
Metrics	211
Tracing	215
Conclusions	217
Part VI. Epilogue	218
Part VII. Appendices	219
Appendix A: Dependency Versions	220
Issue Tracker Web Services	220
Issue Tracker User Interface	220
Appendix B: Iconography	221
Issue Type Icons	221
Issue Priority Icons	221
Miscellaneous Icons	221

CONTENTS

Appendix C: CoffeeScript	222
Appendix D: Gulp	223

Part I. Introduction

Paradigm Shift

In the 1990's savvy marketing and business executives challenged information technology teams to move their mainframe and desktop applications to the Internet. Technology executives soon realized the benefits of a browser-based user interface. Desktop and mainframe user interfaces were tightly coupled to the back-end business logic. Releasing a software update inevitably required changes to the user interface. Desktop user interfaces were often deployed to thousands of client machines. Rolling out updates to non-browser-based user interfaces was tremendously costly and complex. Moving the user interface to the browser nearly eliminated the costly software upgrade process associated with traditional user interfaces.

Throughout the 1990's open-source technologies began emerging that serve as the foundation for web applications. In 1995 Netscape introduced JavaScript which allowed programmers to perform basic tasks such as form input validation or dynamically show and hide portions of HTML pages. In 1999, the Java introduced a Servlet specification revision which ushered in the concept of the server-side web application.

For the next decade, server-side web applications replaced the desktop client as the standard user interface technology. Server-side web applications are characterized by their presentation logic residing on the server that hosts the user interface. When a user performs an action in their browser, the browser submits a request to the server and blocks the user from taking further action. The server processes the request and creates a new HTML page which it returns to the user's browser. The browser displays the new page provided by the server.

In the mid-2000's, Asynchronous JavaScript And XML, or AJAX, technologies had matured and were introduced into many commercial web applications. By the late 2000's and into the 2010's a new trend in web applications emerged, the Single Page Application, or SPA. In a Single Page Application, the presentation layer logic is created entirely in a client-side scripting language like JavaScript. When a user accesses a SPA, their browser loads the application in one HTML page, hence the name Single Page Application. As the user interacts with the application, the client-side script performs the behavioral logic that previously required a round trip to the server to complete. When information needs to be acquired from or submitted to the server, AJAX is used to send the requests to the server asynchronously. Unlike a server-side web application, the page does not block the user from continuing to use the web application. The asynchronous nature of AJAX allows the browser to communicate with the server in the background while the user continues to interact with the web application.

In a Single Page Application, the server-side component is no longer responsible for presentation behavior; however, it still receives HTTP requests and responds to them. The server-side component in a SPA application stack consists of web services that implement the business logic for the resources they manage, decoupling them from a specific client. No longer responsible for presentation logic, these web services are completely reusable throughout the technology ecosystem.

Industry standard technologies are evolving and a new wave of frameworks are emerging that propel the Single Page Application application technology stack to the forefront of web application design paradigms.

About the Book

In this book, two technology frameworks are employed to construct a full stack Single Page Application with RESTful web services. The Spring Framework has been a popular, reliable, and feature rich Java technology for more than a decade. The web services component of the application is constructed using the Spring Boot project, leveraging the core Spring Framework and the Spring Data projects. Readers will create the Single Page Application client-side user interface component using the popular Backbone and Backbone.Marionette JavaScript libraries.

This book is a hands-on tutorial, providing step-by-step instructions beginning with setting up the software projects and finishing with a working application that performs create, read, update, and delete operations on a database. As you progress through the book, you will learn to use various popular client-side libraries including: [Backbone.js](http://backbonejs.org/)¹, [Marionette.js](http://marionettejs.com/)², [Bootstrap](http://getbootstrap.com/)³, and [jQuery](http://jquery.com/)⁴. The examples within the book are authored in the JavaScript language, but the full application is available in [CoffeeScript](http://coffeescript.org/)⁵ as well. As you construct the web services, you will learn to use the core [Spring Framework](http://projects.spring.io/spring-framework/)⁶, [Spring Boot](http://projects.spring.io/spring-boot/)⁷, and [Spring Data](http://projects.spring.io/spring-data/)⁸ for both JPA and MongoDB database persistence. After completing this book, you will be well on your way to becoming a full stack software engineer.

The book focuses on the development of the user interface and web services; however, the *Production Features* section will help you prepare for deployment to servers. The chapters offer solutions for continuous integration, environment-specific configuration, production databases like MySQL and MongoDB, and monitoring and management of the application.

¹<http://backbonejs.org/>

²<http://marionettejs.com/>

³<http://getbootstrap.com/>

⁴<http://jquery.com/>

⁵<http://coffeescript.org/>

⁶<http://projects.spring.io/spring-framework/>

⁷<http://projects.spring.io/spring-boot/>

⁸<http://projects.spring.io/spring-data/>

Intended Audience

This book is intended for software engineers, architects, and enthusiasts. The reader is empowered to create modern, single-page user interface applications and the RESTful web services that serve them data. While reading this book, the concepts are illustrated by building a simple issue tracking application.

The Single Page Application, or SPA, is created using JavaScript, CSS, and HTML. It is recommended that readers have a basic understanding of the JavaScript programming language.

The RESTful web services are created using the Spring Framework in the Java programming language. It is recommended that readers have a basic understanding of the Java programming language.

Git Repositories

Throughout this book there are many code excerpts used to illustrate the concept that is being discussed. The source code for each application component is available on GitHub.

Web Services

The complete web services project is hosted on GitHub in the [issue-tracker-ws-boot-jpa](https://github.com/mwarman/issue-tracker-ws-boot-jpa)⁹ repository.

In the *Production Features* section of this book, the web services are integrated with the MongoDB NoSQL database. While the application changes are subtle, it is best illustrated in a separate hosted repository. The complete web services project integrated with MongoDB is hosted on GitHub in the [issue-tracker-ws-boot-mongodb](https://github.com/mwarman/issue-tracker-ws-boot-mongodb)¹⁰ repository.

Single Page Application

The complete user interface project is hosted on GitHub in the [issue-tracker-ui-marionette-js](https://github.com/mwarman/issue-tracker-ui-marionette-js)¹¹ repository.

Many client-side software engineers prefer CoffeeScript to JavaScript. The complete user interface project authored in CoffeeScript is hosted on GitHub in the [issue-tracker-ui-marionette-coffee](https://github.com/mwarman/issue-tracker-ui-marionette-coffee)¹² repository.

⁹<https://github.com/mwarman/issue-tracker-ws-boot-jpa>

¹⁰<https://github.com/mwarman/issue-tracker-ws-boot-mongodb>

¹¹<https://github.com/mwarman/issue-tracker-ui-marionette-js>

¹²<https://github.com/mwarman/issue-tracker-ui-marionette-coffee>

Part II. Server Foundation

Bootstrap the Server Project

A Proper Foundation

Let's be honest. Starting a new project, setting up your computer is just plain boring. It is one of those tasks that we rush through to get to the fun stuff, writing code.

While it may be boring, painful, and tedious, setting up the project is one of the most important activities you will perform. A good project setup makes building the application a breeze. When you are ready to write code, a good project setup allows you to write a few lines of code, kick off the build, and test those changes with a few keystrokes. During application construction you will perform this set of steps hundreds, perhaps thousands, of times. On the other hand, a bad project setup can make the construction process a laborious, time consuming, chore.

Let's create a proper project foundation and reap the rewards again and again as we build the application.

Installing the Dependencies

The web services project requires two third-party libraries to be installed on your computer, Java and Maven.

The project is written in the Java programming language. The Java Development Kit, or JDK, installation contains the core Java programming language and the Java compiler. You may read more about the JDK on the official [Oracle Java SE website](http://www.oracle.com/technetwork/java/javase)¹³.

Apache Maven is the de facto industry standard tool for Java software project dependency management and build process workflow execution. If you are not familiar with Maven, you may learn more about it on the official [Apache Maven website](http://maven.apache.org/)¹⁴.

Java Development Kit

The Spring Framework serves as the foundation for the web services project. The Spring Framework requires Java 6 or higher. It is strongly recommended that you use the latest version of Java to have the latest security, performance, and functional features available. If the Java Development Kit is already installed on your computer and it meets the minimum requirements, you may skip this section.

¹³<http://www.oracle.com/technetwork/java/javase>

¹⁴<http://maven.apache.org/>

When working in a team environment, it is important that all team members use the same version of Java to work on a given project. Furthermore, it is important that the developers use the same version of Java that is installed on the servers in your operational environments. By ensuring that a consistent version of Java is installed throughout your ecosystem, it eliminates the possibility of bugs arising from the use of different Java versions.

Java is installed differently depending upon the operating system of the computer. The official Oracle Java Standard Edition, or SE, installation instructions describe the installation steps for all major operating systems.

- [Official JDK 8 Installation Guide¹⁵](#)
- [Official JDK 7 Installation Guide¹⁶](#)



Install the JDK

Remember to install the Java Development Kit, or JDK, and not just the Java Runtime Environment, or JRE. The JRE only contains the modules to run Java applications and does not contain the modules necessary to construct or compile Java applications.

After you have installed the JDK on your computer, execute the `java -version` command at a terminal prompt to verify the installation.

Java Version Test

```
$ java -version
```

```
java version "1.7.0_67"
```

```
Java(TM) SE Runtime Environment (build 1.7.0_67-b01)
```

```
Java HotSpot(TM) 64-Bit Server VM (build 24.65-b04, mixed mode)
```

If the Java version is printed, the installation is successful. If not, retrace your steps through the installation guide.

Maven

Apache Maven is used to manage the project dependencies and to compile and package the web services project. A Maven plugin is used to run the web services project on your local machine to create an efficient development environment. If Maven version 3 or higher is already installed on your computer, you may skip this section.

¹⁵http://docs.oracle.com/javase/8/docs/technotes/guides/install/install_overview.html

¹⁶<http://docs.oracle.com/javase/7/docs/webnotes/install/index.html>

Apache Maven is installed differently depending upon the operating system of the computer. First, obtain a stable version from the official [Apache Maven download site](http://maven.apache.org/download.cgi)¹⁷. Then scroll down the page to view the installation instructions for all major operating systems.



Install the JDK First

Maven requires Java. Install the Java Development Kit before installing Maven.

After installing Maven on your computer, execute the `mvn -v` command at a terminal prompt to verify the installation.

Apache Maven Version Test

```
$ mvn -v
```

```
Apache Maven 3.2.3 (33f8c3e1027c3ddde99d3cdebad2656a31e8fdf4; 2014-08-11T16:58:1\
0-04:00)
```

```
Maven home: /usr/lib/maven/latest
```

```
Java version: 1.7.0_67, vendor: Oracle Corporation
```

```
Java home: /usr/lib/jvm/jdk1.7.0_67/jre
```

```
Default locale: en_US, platform encoding: UTF-8
```

```
OS name: "linux", version: "3.2.0-69-generic", arch: "amd64", family: "unix"
```

If the Maven version is printed, the installation is successful. If not, retrace your steps through the installation guide.

Choosing a Source Editor

There are many excellent, open-source Java editors, also known as Integrated Development Environments (IDE), available. Eclipse, NetBeans, and the Spring Tool Suite are some of the most popular IDEs.

The web services application that we construct in this book utilizes the Spring Framework. Therefore, throughout the book, the Spring Tool Suite, or simply STS, is used for the web services project. STS offers many advantages and efficiencies to developers working on Java projects; however, the book does not take shortcuts. You may use a plain text editor for the duration of this book and achieve the same result as if you used STS.

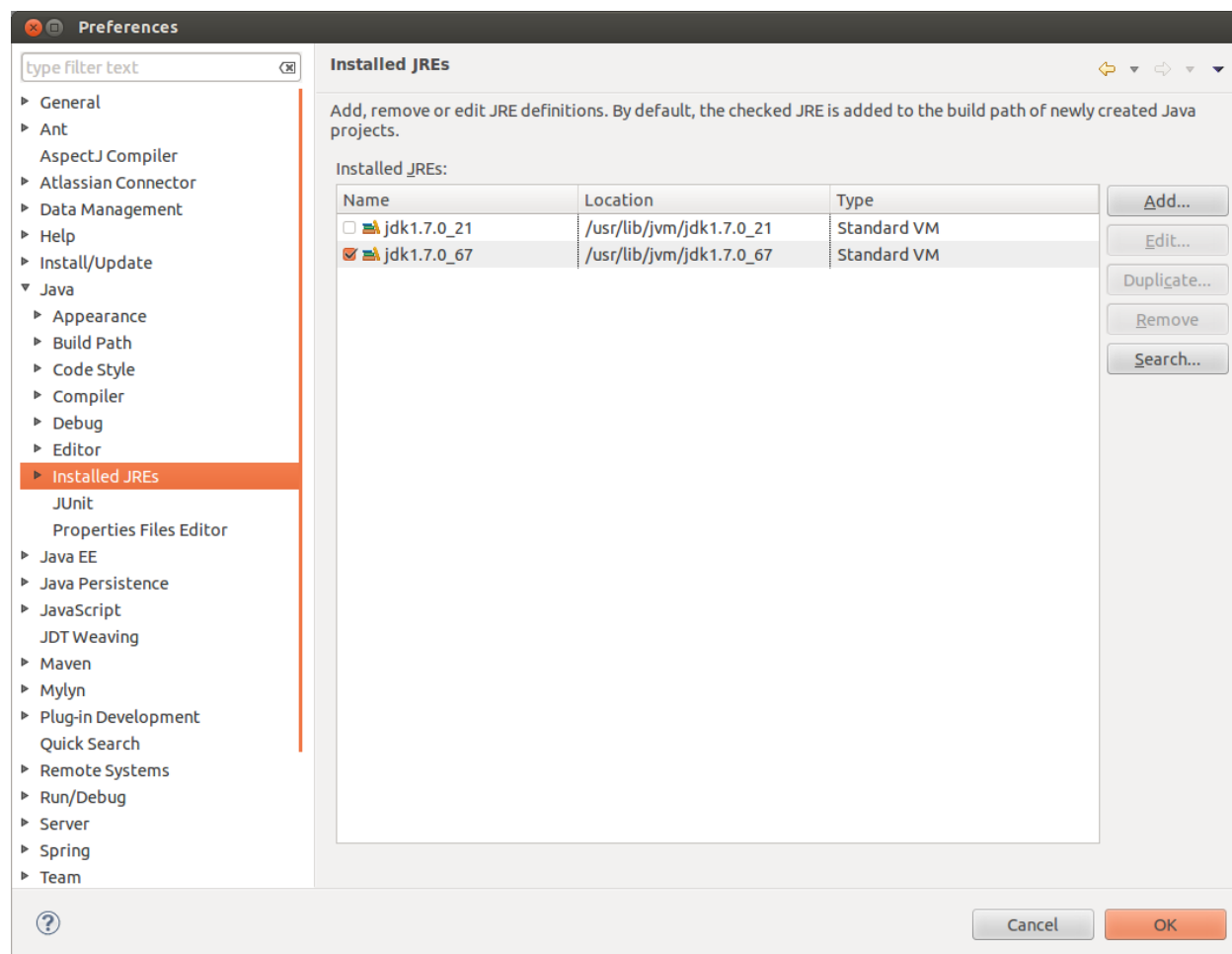
¹⁷<http://maven.apache.org/download.cgi>

Spring Tool Suite

To download and install STS, proceed to the [Spring Tool Suite](http://spring.io/tools/sts)¹⁸ home page, click the download button, and follow the instructions for your operating system. If you choose to use your favorite IDE or a plain text editor, you may skip this section.

After completing the instructions to install STS, click the program icon to start it. The first time you start STS, you are prompted for a location to create a development workspace. The development workspace is where the information about development projects is stored.

After confirming the workspace directory, the STS editor starts. Open the STS preferences window and navigate to the **Installed JREs** section by expanding the **Java** section and next selecting **Installed JREs** as depicted below.

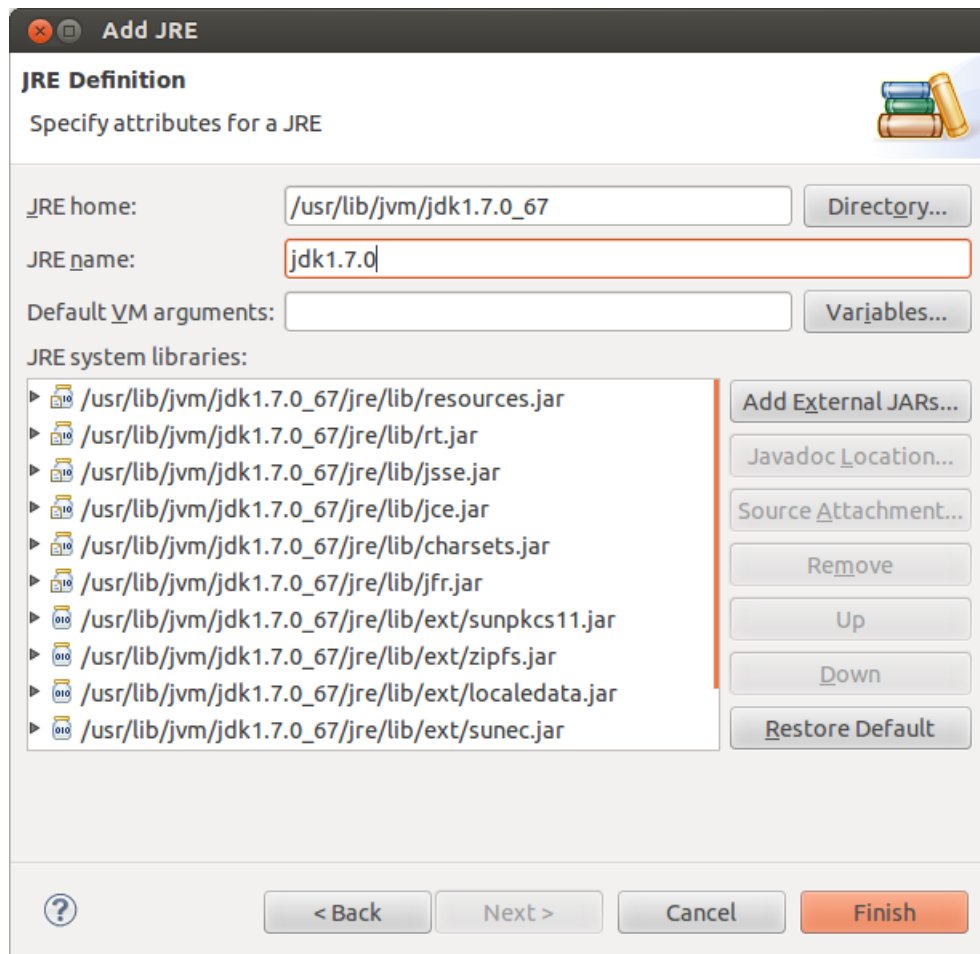


STS Preferences - Installed JREs

Look at the list of installed Java runtimes. If the Java Development Kit, JDK, that you installed previously is selected, no further action is required. If you do not see the JDK that was installed or

¹⁸<http://spring.io/tools/sts>

would like to add a new JDK, click the **Add** button on the **Installed JREs** preference window and select **Standard VM**. Complete the form to add a new Java runtime to STS.



STS Preferences - Add JRE

Once you have entered the JRE home directory and given it a name, click the **Finish** button. On the **Installed JREs** window, check the box next to the Java runtime you wish to use as the default for all new Java projects created in STS.

Structuring the Project

Before we can begin coding web services, we must create the project structure that will contain the source code and configuration. Maven has several built-in project archetypes to facilitate rapid project bootstrapping.

Using Maven Archetypes

A Maven archetype is essentially a script containing instructions for creating a software project template. The archetype tells Maven how to create the directory structure and starter source files for a particular type of application. There are Maven archetypes for J2EE, portlet, web, and other application types.

Since we are using the Spring Framework to construct the web services, we do not need a specialized Maven archetype for the project structure. We will use the *quickstart* archetype. The quickstart archetype creates a basic directory structure for source code, unit tests, and configuration files.

Open a terminal window. At the prompt, issue the Maven archetype command shown below.



Line Wrapping

Omit the backslash `\` characters at the end of each line. They simply indicate that the line is wrapping to fit the width of the book page.

```
$ mvn archetype:generate -DgroupId=org.example.issuetracker -DartifactId=issue-tracker-ws -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

The command invokes the `mvn` Maven executable. The `archetype:generate` portion of the command is called a Maven goal. The goal tells Maven what you wish to do. Following the goal are a collection of command line arguments prefixed by `-D`. These arguments are passed into Maven and provide configuration to the `archetype:generate` goal.

The `groupId` argument is the fully qualified base package name for your project. The archetype uses the supplied `groupId` to create the project structure and inserts it into the starter `pom.xml`. Often, but not always, organizations use the same `groupId` value for all projects.

The `artifactId` argument is the unique name of an artifact within the group. The archetype uses the supplied `artifactId` as the base directory name for the project and inserts it into the starter `pom.xml`.

The `archetypeArtifactId` argument value tells Maven which type of project template to use when generating the project structure.

The `interactiveMode` argument value of *false* instructs Maven to use the supplied parameters to generate the project structure without prompting for confirmation. If this parameter is omitted, Maven prompts for values for any parameters the archetype requires or to confirm the values that are supplied via `-D` parameters.

After submitting the command, Maven downloads the dependencies required to execute the archetype and generate the project structure. If this is the first execution of a Maven archetype generation on the computer, more downloads will occur. The command output is similar to that which is depicted below.

Console Log

```

[INFO] Scanning for projects...
Downloading: http://.../repo/org/apache/maven/plugins/maven-install-plugin/2.4/m\
aven-install-plugin-2.4.pom
Downloaded: http://.../repo/org/apache/maven/plugins/maven-install-plugin/2.4/ma\
ven-install-plugin-2.4.pom (7 KB at 1.3 KB/sec)
[INFO]
[INFO] -----
[INFO] Building Maven Stub Project (No POM) 1
[INFO] -----
[INFO]
[INFO] >>> maven-archetype-plugin:2.2:generate (default-cli) > generate-sources \
@ standalone-pom >>>
[INFO]
[INFO] <<< maven-archetype-plugin:2.2:generate (default-cli) < generate-sources \
@ standalone-pom <<<
[INFO]
[INFO] --- maven-archetype-plugin:2.2:generate (default-cli) @ standalone-pom ---
[INFO] Generating project in Batch mode
[INFO] -----\
---
[INFO] Using following parameters for creating project from Old (1.x) Archetype:\
maven-archetype-quickstart:1.0
[INFO] -----\
---
[INFO] Parameter: groupId, Value: org.example.issuetracker
[INFO] Parameter: packageName, Value: org.example.issuetracker
[INFO] Parameter: package, Value: org.example.issuetracker
[INFO] Parameter: artifactId, Value: issue-tracker-ws
[INFO] Parameter: basedir, Value: /home/user/projects
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] project created from Old (1.x) Archetype in dir: /home/user/projects/issu\
e-tracker-ws
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 26.148 s
[INFO] Finished at: 2014-09-28T11:04:29-04:00
[INFO] Final Memory: 14M/361M
[INFO] -----

```

Remember that we used `artifactId=issue-tracker-ws` which Maven uses to create the project

base directory. The generated project structure looks like this:

```

issue-tracker-ws
├── pom.xml
└── src
    ├── main
    │   ├── java
    │   │   ├── org
    │   │   │   ├── example
    │   │   │   │   ├── issuetracker
    │   │   │   │   │   └── App.java
    │   └── test
    │       ├── java
    │       │   ├── org
    │       │   │   ├── example
    │       │   │   │   ├── issuetracker
    │       │   │   │   │   └── AppTest.java

```

Congratulations! You have successfully bootstrapped the web services project. Next, we will import the project into Spring Tool Suite and modify the starter pom.xml and source files to construct the first web service.

Importing the Project

Let's import the web services project into the Spring Tool Suite (STS) IDE. The STS plugin which manages Maven projects requires a small change to the pom.xml file before importing the project into STS.

Updating the POM

First, let's look at the 'pom.xml' file generated by the Maven quickstart archetype.

Generated Maven pom.xml File

```

1  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/\
2  2001/XMLSchema-instance"
3     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/\
4  maven-v4_0_0.xsd">
5     <modelVersion>4.0.0</modelVersion>
6     <groupId>org.example.issuetracker</groupId>
7     <artifactId>issuetracker-ws</artifactId>
8     <packaging>jar</packaging>

```

```
9     <version>1.0.0-SNAPSHOT</version>
10    <name>issuetracker-ws</name>
11    <url>http://maven.apache.org</url>
12    <dependencies>
13      <dependency>
14        <groupId>junit</groupId>
15        <artifactId>junit</artifactId>
16        <version>3.8.1</version>
17        <scope>test</scope>
18      </dependency>
19    </dependencies>
20  </project>
```

Lines 6 and 7 contain the project groupId and artifactId values which were supplied to the archetype command. Line 9 contains the default version value.

Lines 12 through 19 contain the project dependencies. The quickstart archetype generates two Java classes: a main class named App.java and a JUnit test class named AppTest.java. The JUnit test depends upon the JUnit framework library to compile and run, therefore it is declared in the project dependencies section of the POM.

Before importing the project into STS, we need to make a few changes to the POM. Open the 'pom.xml' file in a text editor.

Updated Maven pom.xml File

```
1  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/\
2  2001/XMLSchema-instance"
3    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/\
4  maven-v4_0_0.xsd">
5
6    <modelVersion>4.0.0</modelVersion>
7
8    <groupId>org.example.issuetracker</groupId>
9    <artifactId>issuetracker-ws</artifactId>
10   <version>1.0.0-SNAPSHOT</version>
11   <packaging>jar</packaging>
12   <name>issuetracker-ws</name>
13   <url>http://maven.apache.org</url>
14
15   <properties>
16     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
17     <java.version>1.7</java.version>
18   </properties>
```

```
19
20     <dependencies>
21         <dependency>
22             <groupId>junit</groupId>
23             <artifactId>junit</artifactId>
24             <version>3.8.1</version>
25             <scope>test</scope>
26         </dependency>
27     </dependencies>
28
29     <build>
30         <plugins>
31             <plugin>
32                 <groupId>org.apache.maven.plugins</groupId>
33                 <artifactId>maven-compiler-plugin</artifactId>
34                 <version>3.1</version>
35                 <configuration>
36                     <source>${java.version}</source>
37                     <target>${java.version}</target>
38                 </configuration>
39             </plugin>
40         </plugins>
41     </build>
42
43 </project>
```

To improve the readability of the POM, some whitespace has been added to separate the logical sections of the file.

Lines 15 through 18 introduce a new section to the POM, the project properties. The `properties` section contains configuration for the Maven project. The `properties` element contains child elements whose element name is the name of the configuration property and whose element value is the configuration property value.

The `project.build.sourceEncoding` property informs Maven to expect UTF-8 encoded source files and, where applicable, to produce UTF-8 encoded files.

The `java.version` property is not directly used by Maven. Instead, this property value is used to replace property placeholder references located throughout the POM. Lines 36 and 37 contain property placeholder references for the `java.version` property. Property placeholder references are prefixed with a dollar sign followed by curly braces containing a property name such as `${property.name}`. When the Maven process is started, property placeholder references are located and replaced with actual property values.

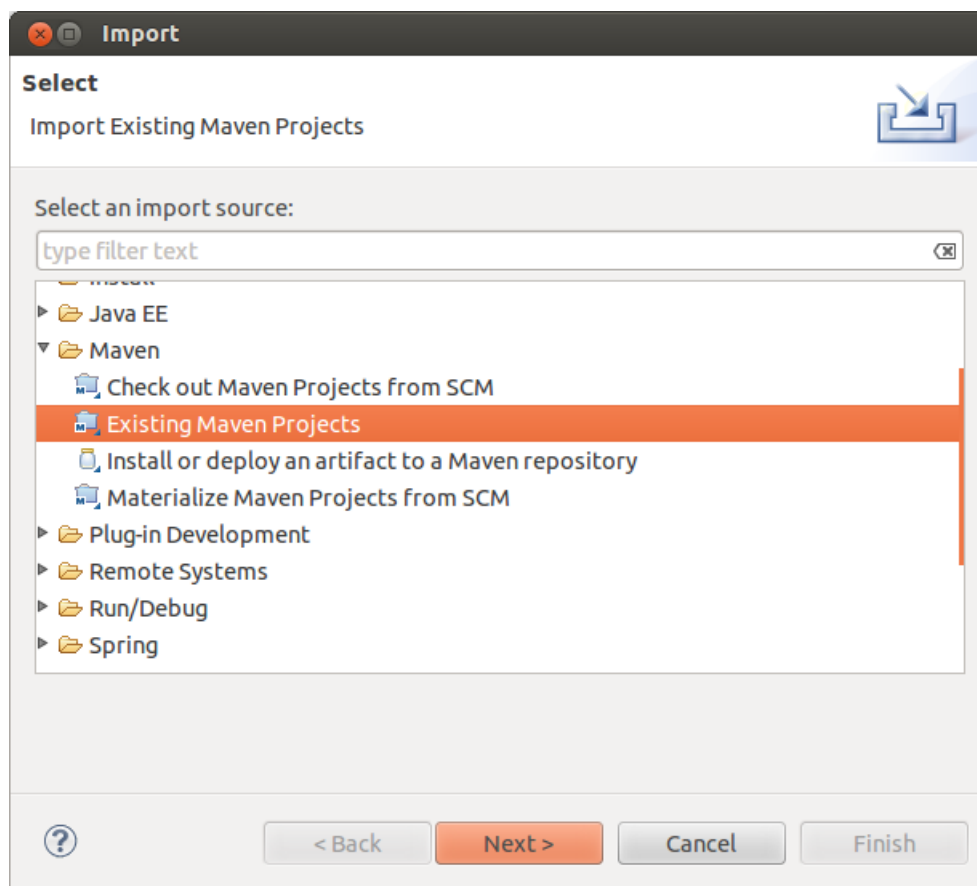
Lines 29 through 41 introduce a new section to the POM, the build section. The build section contains configuration and plugins that enhance or augment Maven's capabilities. The Maven compiler plugin is added to the POM to override the default Java compiler version used by Maven, 1.5. In this example, we instruct Maven to override the default and use version 1.7 of the Java compiler.

Java Versions

Java version 1.7 is the same as Java 7. Java version 1.8 is the same as Java 8. If you are using Java 8 on your development computer, set the `java.version` to 1.8 in the properties section of your POM.

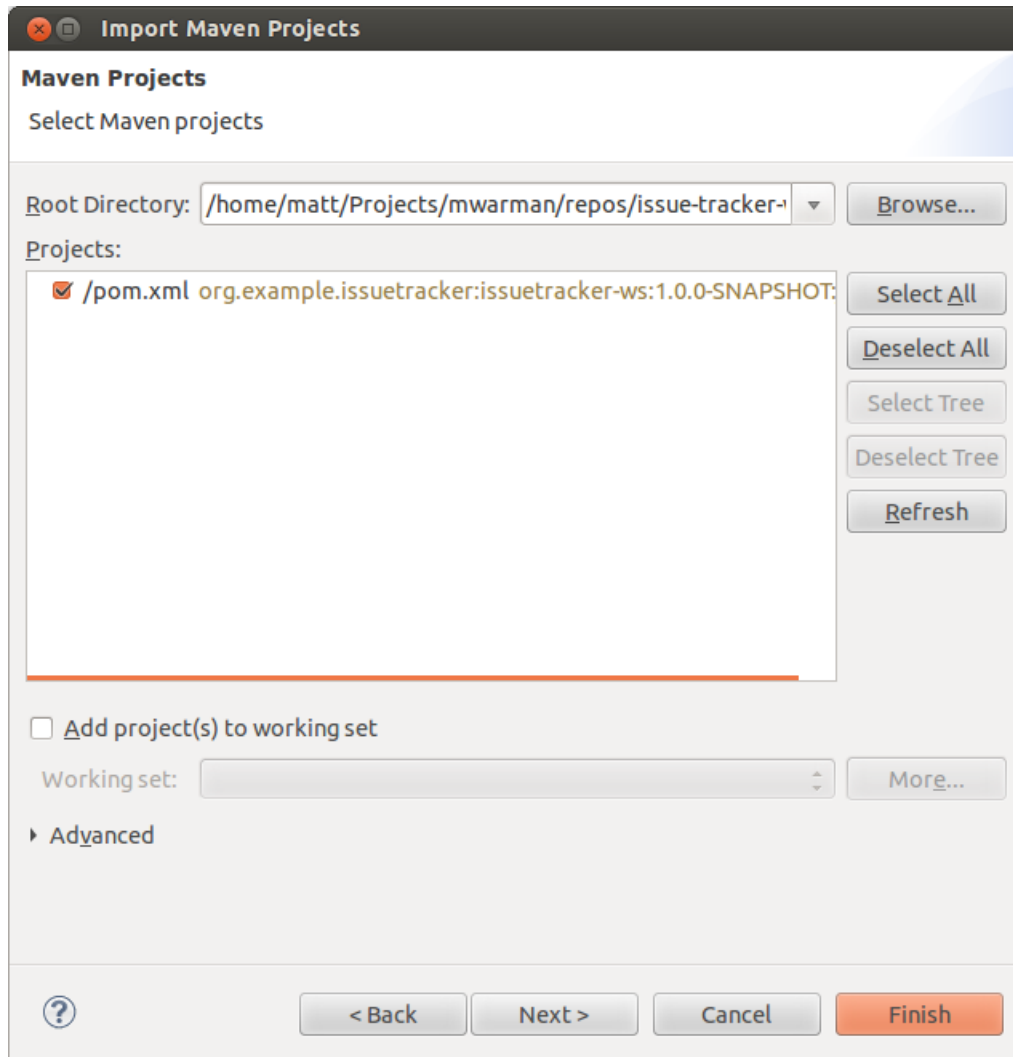
Creating the STS Project

Start the Spring Tool Suite application. From the menu bar, open the *File* menu and click *Import*. STS opens the Import window depicted below. Expand the *Maven* section and select Existing Maven Projects.



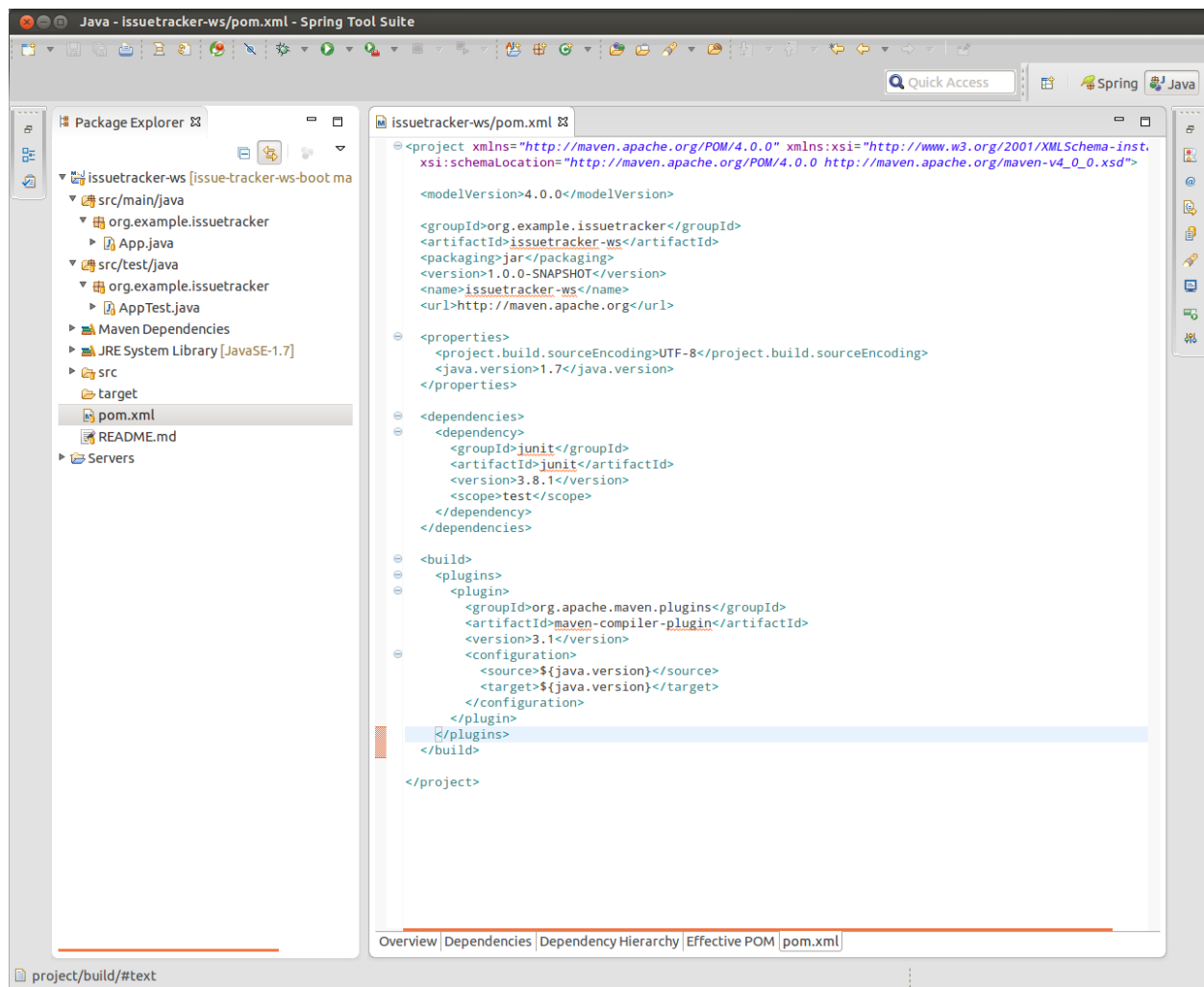
Import - Existing Maven Projects

Click the *Next* button to open the Import Maven Projects window depicted below. Click the *Browse* button and select the base directory of the issue tracker web services project. This is the directory where the `pom.xml` file is located.



Import Maven Projects

Click the *Finish* button to import the project into Spring Tool Suite. Once imported, your STS workspace will look similar to the screenshot below.



STS Workspace - Java Perspective



The Road Ahead

In the next chapter, Spring Boot is introduced and integrated into the web services project.

Getting Started with Spring Boot

The Spring Boot project enables software engineers to rapidly build applications. Applications built atop Spring Boot may begin as prototypes, leveraging Spring Boot's built-in capability to automatically configure the project. As the application shifts from prototype to production, it is simple to replace Spring Boot's inferred configuration with explicit instrumentation.

Engineers that have used the Spring Framework previously may hesitate to use Spring Boot, feeling that some degree of control over configuration is relinquished by using Spring Boot. That is not the case. One of the most powerful features of Spring Boot is the ability to automatically configure Spring by examining the project dependencies. Engineers can rapidly prototype and demonstrate working application features to their customers or, perhaps more importantly, their bosses. However, Spring Boot seamlessly moves aside when engineers add specific configuration values that replace the automatic configuration. Furthermore, Spring features that are enabled by default may be easily disabled through configuration.



The project in this book uses Spring Boot; however, it could easily be created using the Spring projects themselves without Spring Boot.

Updating the POM

To add Spring Boot to the web services project, let's edit the Maven POM file, `pom.xml`. The revised POM file is depicted below followed by an explanation of the changes.

Maven POM with Spring Boot

```
1 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/\
2 2001/XMLSchema-instance"
3   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/\
4 maven-v4_0_0.xsd">
5
6   <modelVersion>4.0.0</modelVersion>
7
8   <groupId>org.example.issuetracker</groupId>
9   <artifactId>issuetracker-ws</artifactId>
10  <version>1.0.0-SNAPSHOT</version>
11  —<packaging>jar</packaging>
12  —<name>issuetracker-ws</name>
```

```
13 —<url>http://maven.apache.org</url>
14
15   <parent>
16     <groupId>org.springframework.boot</groupId>
17     <artifactId>spring-boot-starter-parent</artifactId>
18     <version>1.2.4.RELEASE</version>
19   </parent>
20
21   <properties>
22     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
23     <java.version>1.7</java.version>
24   </properties>
25
26   <dependencies>
27     <dependency>
28       <groupId>org.springframework.boot</groupId>
29       <artifactId>spring-boot-starter-web</artifactId>
30     </dependency>
31     <dependency>
32       <groupId>org.springframework.boot</groupId>
33       <artifactId>spring-boot-starter-test</artifactId>
34       <scope>test</scope>
35     </dependency>
36   </dependencies>
37
38   <build>
39     <plugins>
40       <plugin>
41         <groupId>org.springframework.boot</groupId>
42         <artifactId>spring-boot-maven-plugin</artifactId>
43       </plugin>
44     </plugins>
45   </build>
46
47 </project>
```

The `url` and `name` elements have been removed. Their purpose is to provide additional description if the project is published to a Maven repository, such as [Maven Central](http://central.sonatype.org/)¹⁹.

The `packaging` element has been removed. This element instructs Maven how to package the build artifact. The Spring Boot Maven plugin provides packaging features that make the packaging

¹⁹<http://central.sonatype.org/>

element obsolete.

The `parent` section has been added. This section declares the Spring Boot Starter POM to be the parent of our POM file, allowing our POM to inherit some helpful configuration.

The `dependencies` section has been modified. The Spring Boot project offers several starter dependency packages. One of those is the `spring-boot-starter-web` dependency which includes a bundle of libraries needed to construct web applications. The JUnit dependency has been replaced with the `spring-boot-starter-test` dependency, which declares a transitive dependency on JUnit in addition to other relevant test libraries. Note that the new dependencies omit the `version` element. By inheriting dependency configuration from the Spring Boot Starter POM, the versions of certain artifacts may be omitted because Spring Boot ensures that the latest supported version is installed.

The `build` section has been modified. The plugin for the Maven compiler has been replaced with the `spring-boot-maven-plugin`. This plugin provides new Maven functionality for packaging and running Spring Boot projects.

Application.java

Spring Boot applications may be packaged as a JAR file and executed from the command line using `java -jar application.jar` where `application.jar` is the name of the Spring Boot application. The `Application` class serves several important functions in a Spring Boot application. It implements the standard Java `main(String... args)` method, serving as the runtime entry point for the application. The `Application` class has annotations that provide configuration for Spring Boot and the Spring `ApplicationContext`. Let's begin by modifying the `App.java` source file generated by the Maven archetype, transforming it into the `Spring Boot Application.java` class.

Application.java

```
1 package org.example.issuetracker;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6 /**
7  * Spring Boot main application
8  */
9 @SpringBootApplication
10 public class Application
11 {
12     public static void main( String[] args ) throws Exception
13     {
14         SpringApplication.run(Application.class, args);
15     }
16 }
```

Spring Boot doesn't care about the name of the primary application class. Naming the class `MyApplication`, `IssueTrackerApplication`, or `Foo` would not impact Spring Boot's operation. Let's sacrifice brevity for clarity and rename the source file (and class) to `Application`.

@SpringBootApplication

The `@SpringBootApplication` annotation equates to using the `@Configuration`, `@EnableAutoConfiguration`, and `@ComponentScan` annotations with their default values.

The annotation instructs Spring Boot to examine the project dependencies and determine how to configure the Spring application. We added the `spring-boot-starter-web` dependency to our POM. Due to the presence of that dependency, Spring Boot's auto-configuration adds Spring MVC and Apache Tomcat to the project because it assumes the project is a web application. The Spring `ApplicationContext` is configured with sensible default values appropriate for a Spring MVC project. The embedded Apache Tomcat instance will also be configured with default values.

The `@SpringBootApplication` annotation instructs Spring to scan the application for stereotype annotations such as `@Component`, `@Repository`, `@Service`, `@RestController`, and `@Configuration`. When those components are identified by the scanner, they are registered in the `ApplicationContext`.

Like the `@Configuration` annotation, `@SpringBootApplication` tells Spring that the `Application` class may supply additional configuration to the application.

The default configuration applied by the `@SpringBootApplication` may be overridden in a variety of ways. The configuration may be "tuned" through the use of additional annotations or a properties file. It is also possible to enable or disable entire suites of Spring functionality through the application of specific annotations. The [Configuration](#) chapter discusses Spring Boot project configuration in detail.

The "main" Method

The current `main` method simply prints "Hello World!". Let's replace that line with something a bit more useful for our purposes.

The `main` method begins the execution of the `SpringApplication` class by invoking its `run` method. The `SpringApplication` class starts a Spring `ApplicationContext` hosted within the embedded Apache Tomcat web application server.

The `SpringApplication.run` method accepts two parameters. The first is the name of the primary Spring component. In our case, it is this class. The second parameter is the `args` variable which contains any command line arguments supplied when the application was initiated.

Starting the Server

That's it! You have written all of the code necessary to create a running Spring MVC web application using Spring Boot. Let's run the application. There are several ways to start the application. We will

discuss two: Maven and executable JARs. You, or your organization, may prefer to generate a WAR file and deploy that to a standalone web application server like Apache Tomcat or Jetty. The Spring Boot Maven plugin is capable of generating a WAR file. If you are interested in WAR packaging, more information is available in the [Spring Boot Reference Guide](#)²⁰.

Running with Maven

Open a terminal and navigate to the base project directory where the `pom.xml` file is located. Type `mvn spring-boot:run` and press *Enter*.

Starting the Spring Boot Application

```

1  $ mvn spring-boot:run
2
3  Scanning for projects...
4
5  -----
6  Building issuetracker-ws 1.0.0-SNAPSHOT
7  -----
8
9
10  .      _ _ _ _ _
11  /\ \  / ___ ' _ _ _ _ _ ( _ ) _ _ _ _ _ \ \ \ \
12  ( ( ) \___ | ' _ | ' _ | ' _ \ _ ` | \ \ \ \
13  \ \ / ___ | | _ | | | | | | | ( _ | | ) ) ) )
14  '  | ___ | . _ | | | _ | | \ _ , | / / / /
15  =====|_|=====|___/=/_/_/_/
16  :: Spring Boot ::                (v1.2.4.RELEASE)
17
18  (excluded log output)
19
20  s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080/http
20  org.example.issuetracker.Application      : Started Application in 2.525 seconds

```

The project POM file declares the Spring Boot Starter POM as its parent and includes the Spring Boot Maven plugin. One of the features made available through the Spring Boot Maven plugin is the ability to run the application using Maven. This is not a practical way to run the application in a server environment; however, it is very useful for engineers to perform iterative development on their computers.

Near the end of the console output words similar to “Started Application in n.nnn seconds” indicates that the embedded Apache Tomcat container has started and is hosting the application. Open a

²⁰<http://docs.spring.io/spring-boot/docs/1.2.4.RELEASE/reference/htmlsingle/#build-tool-plugins-maven-packaging>

browser and type `http://localhost:8080/` into the address bar. An error page is shown. This is actually a successful test! We have yet to create any functionality in the application so we are served the default error page. The Apache Tomcat web server and Spring Boot application performed precisely as expected.

To stop the application at any time, press `ctrl-c`. The Spring application will stop cleanly, publishing all of the normal `ApplicationContext` shutdown events.

Creating an Executable JAR

It is impractical and unreliable to use Maven to run the application in production, or non-production server environments. The Spring Boot Maven plugin offers a second packaging option, an executable JAR file.

Open a terminal and navigate to the base project directory where the `pom.xml` file is located. Type `mvn clean package` and press *Enter*.

Creating an Executable JAR

```
1 $ mvn clean package
2
3 -----
4 Building issuetracker-ws 1.0.0-SNAPSHOT
5 -----
6
7 (excluded log output)
8
9 Building jar: issue-tracker-ws-boot/target/issuetracker-ws-1.0.0-SNAPSHOT.jar
10
11 --- spring-boot-maven-plugin:1.2.4.RELEASE:repackage (default) @ issuetracker-ws
12 -----
13 BUILD SUCCESS
14 -----
```

Maven performs the standard lifecycle operations to clean and package the artifact for the project. We have omitted the `package` element from our POM, so the default artifact type of JAR is used. On line 11 of the console output, note that the Spring Boot Maven plugin `repackage` goal is executed automatically, ensuring that the JAR file produced contains the embedded Apache Tomcat server and other necessary dependencies so that it may be executed from the command line.

To run the executable jar, use the `java -jar` command depicted below.

Running an Executable JAR

```

1  $ java -jar target/issuetracker-ws-1.0.0-SNAPSHOT.jar
2
3
4  .   _ _ _ _ _
5  /\ \ / _ _ ' _ _ _ _ _ ( _ ) _ _ _ _ _ \ \ \ \
6  ( ( ) \ _ _ | ' _ | ' _ | ' _ \ / _ ` | \ \ \ \
7  \ \ / _ _ | | _ | | | | | | | ( _ | | ) ) ) )
8  '   | _ _ | . _ | | | | | _ \ _ , | / / / /
9  =====|_|=====| _ _/=/_/_/_/
10 :: Spring Boot ::                (v1.2.4.RELEASE)
11
12 org.example.issuetracker.Application      : Starting Application with PID 1917
13 (excluded log output)
14
15 s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080/http
16 org.example.issuetracker.Application      : Started Application in 3.011 seconds

```

When deploying to a server environment, an executable JAR file is very practical. Deploying a traditional WAR file to a standalone web server involves several steps such as: stop the web server; clean up the previously deployed application; deploy the updated application; and restart the web server. Deploying an executable Spring Boot application JAR requires only replacing the JAR file on the server and restarting the process. The [Development Operations](#) chapter discusses server hosting approaches.

To stop the application at any time, press `ctrl-c`. The Spring application will stop cleanly, publishing all of the normal `ApplicationContext` shutdown events.



The Road Ahead

In the next chapter we explore the capabilities of Spring Boot by constructing the first RESTful web service.

Building the First RESTful Web Service

Cohesion

Cohesion is a relative measure of the degree to which the related pieces of an application module are grouped. In software design and development it is desirable for modules to have a high cohesion, meaning that the application modules are organized into groups that “*belong*” together. Often, the term cohesion is described as the “separation of concerns”, again referring to the practice of grouping common functionality or behaviors together. Software modules that are highly cohesive tend to be simpler to understand, easier to maintain, and have a higher degree of reusability than modules with low cohesion.

The principles of cohesion may be applied to many levels of granularity within an application. In this chapter, as we begin to construct the web services, you will see that the application is organized into several logically cohesive packages including: the web service controllers; the entity model; the business services; and the data repository. Within those packages, common functionality is organized into distinct classes, further increasing the cohesion of our application.

Updating the POM

Initially, the issue tracker application uses a [HSQLDB](http://hsqldb.org/)²¹ database. HSQLDB is a SQL relational database written in Java and, by default, stores it’s data in-memory. In later chapters, we will learn how to replace HSQLDB with the MySQL relational database and the MongoDB NoSQL database.

Since HSQLDB is a relational database, we will use the [Spring Data JPA](http://projects.spring.io/spring-data-jpa/)²² project to serve as the data access layer of the application. The Spring Data JPA project implements all boilerplate code typically required for interaction with relational databases. When the application is converted to use MySQL instead of HSQLDB, only configuration changes are required because Spring Data JPA abstracts the persistence implementation from the application logic.

Let’s add two dependencies to the Maven POM file, `pom.xml`, to support JPA database persistence with HSQLDB.

²¹<http://hsqldb.org/>

²²<http://projects.spring.io/spring-data-jpa/>

Updated Dependencies

```
<dependencies>

...

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
  <groupId>org.hsqldb</groupId>
  <artifactId>hsqldb</artifactId>
  <scope>runtime</scope>
</dependency>

...

</dependencies>
```

In the dependencies section of the `pom.xml` file, add `spring-boot-starter-data-jpa` and `hsqldb` as depicted above. Note that, again, we did not need to supply the version for these dependencies. Both Spring Data JPA and HSQLDB are recognized by the Spring Boot Starter Parent POM and the latest supported version is automatically included in our project. The next time the application is started, Spring Boot will detect the presence of these new dependencies on the classpath and automatically search the project classes for JPA repositories and entities. In the next few sections, we will elaborate further on JPA repositories and entities while constructing the first web service.

Creating the Entity Model

The first step when constructing any application, is to design and construct the data model. The issue tracker application has a relatively simple data model which centers around one main entity type, the *Issue*. The Issue entity has a title, description, type, priority, and status. The title and description are defined as String values. The type, priority, and status attributes each may contain a specific set of constant values and, therefore, are defined as Enums.

Issue Enums

Let's begin by creating the three Enum types: `IssueType`, `IssuePriority`, and `IssueStatus`. First, create a package named `org.example.issuetracker.model` within the main application code structure, `src/main/java`. Within the new `model` package, create the three Enum types depicted below.

IssueType Enum

```
1 package org.example.issuetracker.model;
2
3 public enum IssueType {
4     BUG,
5     ENHANCEMENT,
6     STORY,
7     TASK
8
9 }
```

The IssueType Enum defines the types of Issues that the application supports.

IssuePriority Enum

```
1 package org.example.issuetracker.model;
2
3 public enum IssuePriority {
4     LOW,
5     MEDIUM,
6     HIGH
7
8 }
```

The IssuePriority Enum defines the relative importance or urgency of an Issue.

IssueStatus Enum

```
1 package org.example.issuetracker.model;
2
3 public enum IssueStatus {
4     OPEN,
5     IN_PROGRESS,
6     DONE
7
8 }
```

The IssueStatus Enum defines the workflow or lifecycle of Issues.

The Issue Entity

JPA entity model classes are simply a Plain Old Java Object, or POJO, with annotations that enrich the class with relational persistence metadata. Within the `org.example.issuetracker.model` package, create a new class named `Issue`. The code below defines the attributes of the `Issue` class and introduces some JPA annotations.

Issue Entity

```
1  @Entity
2  public class Issue {
3
4      @Id
5      @GeneratedValue
6      private Long id;
7
8      private String title;
9
10     private String description;
11
12     @Enumerated(EnumType.STRING)
13     private IssueType type;
14
15     @Enumerated(EnumType.STRING)
16     private IssuePriority priority;
17
18     @Enumerated(EnumType.STRING)
19     private IssueStatus status;
20
21     public Issue() {
22
23     }
24
25     // ...
26
27 }
```

@Entity

The `@Entity` annotation is a class-level stereotype annotation that declares this class to be a JPA entity model. When the application starts, a Spring Data JPA *component scanner* searches for classes with `@Entity` annotations. Upon discovering one of these annotated classes, the component scanner registers the class making Spring aware of its existence. When the issue tracker application starts, the component scanner registers the `Issue` class as a JPA persistent entity.

@Id and @GeneratedValue

The `@Id` annotation is an attribute-level annotation. This annotation is declared at most once in an entity class. The `@Id` annotation indicates which attribute of the class is the primary key identifier for the entity.

The `@GeneratedValue` annotation is an attribute-level annotation that may optionally accompany the `@Id` attribute. This attribute tells the JPA framework that the database should automatically generate a value for the primary key identifier attribute when a new entity of this type is stored in the database.

The `Issue` entity declares the `id` attribute as:

```
@Id
@GeneratedValue
private Long id;
```

Analyzing the annotations and the `Long` data type, the JPA framework knows that this attribute is the primary key identifier for this entity; that the attribute type is numeric; and that the database is responsible for generating the attribute values.

@Enumerated

The `@Enumerated` annotation is an attribute-level annotation. This annotation indicates that the attribute value is one from an enumerated type. The annotation is optionally configured with an `EnumType` which informs the JPA framework how to persist the attribute value. If the `EnumType` element is omitted from `@Enumerated`, the JPA framework stores an integer ordinal value representing the location of the specific enumerated value within the set of possible values.

In the `Issue` class, we specify `EnumType.STRING` which instructs the JPA framework to persist the value as the name of the enumerated value. For example, the `IssuePriority` enum contains values `IssueType.LOW`, `IssueType.MEDIUM`, and `IssueType.HIGH`. Using `EnumType.STRING`, the JPA framework stores `String` values `LOW`, `MEDIUM`, or `HIGH` into the `priority` column of the `Issue` table.

The Full Issue Entity

The complete `Issue` entity model class is depicted below.

Complete Issue Entity Class

```
1 package org.example.issuetracker.model;
2
3 import javax.persistence.Entity;
4 import javax.persistence.EnumType;
5 import javax.persistence.Enumerated;
6 import javax.persistence.GeneratedValue;
7 import javax.persistence.Id;
8
9
10 @Entity
11 public class Issue {
12
13     @Id
14     @GeneratedValue
15     private Long id;
16
17     private String title;
18
19     private String description;
20
21     @Enumerated(EnumType.STRING)
22     private IssueType type;
23
24     @Enumerated(EnumType.STRING)
25     private IssuePriority priority;
26
27     @Enumerated(EnumType.STRING)
28     private IssueStatus status;
29
30     public Issue() {
31
32     }
33
34     public Long getId() {
35         return id;
36     }
37
38     public void setId(Long id) {
39         this.id = id;
40     }
41
```



```
42     public String getTitle() {
43         return title;
44     }
45
46     public void setTitle(String title) {
47         this.title = title;
48     }
49
50     public String getDescription() {
51         return description;
52     }
53
54     public void setDescription(String description) {
55         this.description = description;
56     }
57
58     public IssueType getType() {
59         return type;
60     }
61
62     public void setType(IssueType type) {
63         this.type = type;
64     }
65
66     public IssuePriority getPriority() {
67         return priority;
68     }
69
70     public void setPriority(IssuePriority priority) {
71         this.priority = priority;
72     }
73
74     public IssueStatus getStatus() {
75         return status;
76     }
77
78     public void setStatus(IssueStatus status) {
79         this.status = status;
80     }
81
82 }
```

Creating the Repository

A Spring Data Repository eliminates the need for the software engineer to write boilerplate code for every database transaction. To interact with a database in typical code you might need to get a connection, create a query, supply query parameter values, run the query, iterate over the result set, and close the connection. While using an ORM solution like Hibernate removes some of those concerns, you are still obliged to write boilerplate code.

As you will see, Spring Data Repositories reduce the data access layer code to a mere Java interface. Let's create an `IssueRepository` interface to serve as the data access layer for the `Issue` entity. First, create a package named `org.example.issuetracker.repository` within the main application code structure, `src/main/java`. Within the new package, create the `IssueRepository` interface illustrated below.

IssueRepository Interface

```
1 package org.example.issuetracker.repository;
2
3 import org.example.issuetracker.model.Issue;
4 import org.springframework.data.jpa.repository.JpaRepository;
5 import org.springframework.stereotype.Repository;
6
7 @Repository
8 public interface IssueRepository extends JpaRepository<Issue, Long> {
9
10 }
```

This one interface represents the entire data access layer for `Issue` entities. The goal of the Spring Data project is to simplify data access so that projects need only to implement custom persistence requirements. At runtime, Spring Data provides an implementation class, typically `SimpleJpaRepository` that contains the actual data access layer logic for the repository interface.

JpaRepository Interface

The Spring Data project is the umbrella for several implementation specific sub-projects. The Spring Data JPA project is one such sub-project. The core Spring Data project provides the reference architecture which the sub-projects implement and extend.

The Spring Data project abstracts interaction with relational databases through behaviors defined in the `Repository`²³ interface. Furthermore, the project defines two specializations of the `Repository`

²³<http://docs.spring.io/spring-data/data-commons/docs/current/api/org/springframework/data/repository/Repository.html>

interface. The first is `CrudRepository`²⁴ and the second which further specializes `CrudRepository` is `PagingAndSortingRepository`²⁵.

The `CrudRepository` interface defines all of the typical operations performed on an entity class such as `save`, `delete`, `findOne`, `findAll`, and others. Not surprisingly, the aptly named `CrudRepository` provides all CRUD (Create, Read, Update, and Delete) behaviors on an entity class.

The `PagingAndSortingRepository` extends `CrudRepository` providing additional behaviors to find collections of an entity class in smaller subsets, or pages, and provides the means to sort the results.

The `JpaRepository` interface is defined within the Spring Data JPA sub-project. The `JpaRepository` extends `PagingAndSortingRepository` and, therefore, possesses all of the behaviors to perform basic CRUD operations on an entity class as well as provide pagable and sorted results. The reason the `IssueRepository` interface does not define any behaviors is because our application leverages the behaviors made available by `CrudRepository`.

@Repository

The `@Repository` annotation is a class-level stereotype annotation that declares this class to be a repository definition. When a Spring application starts, a *component scanner* searches for classes with stereotype annotations. Upon discovering one of these stereotypes, the component scanner registers the class making Spring aware of its existence. When our application starts, the component scanner will register the `IssueRepository` interface as a JPA repository because it extends the `JpaRepository` interface.

Creating the Service

Business services encapsulate logic, calculations, and data manipulation functionality. Spring provides the `@Service` stereotype to identify service components and enable them for dependency injection. Business services should always define an interface which serves as a client contract, defining the public behaviors made available by the service. The service interface, not the service implementation, is declared in client components using Spring's `@Autowired` annotation, transparently making a fully instantiated and configured service implementation available for use within client classes.

Let's create an `IssueService` interface and `IssueServiceBean` class to serve as the business service layer for activities related to the `Issue` entity. First, create a package named `org.example.issuetracker.service` within the main application code structure, `src/main/java`. Within the new service package, create the `IssueService` interface and `IssueServiceBean` class illustrated below.

²⁴<http://docs.spring.io/spring-data/data-commons/docs/current/api/org/springframework/data/repository/CrudRepository.html>

²⁵<http://docs.spring.io/spring-data/data-commons/docs/current/api/org/springframework/data/repository/PagingAndSortingRepository.html>

Creating the Interface

A Java interface defines the public behaviors, or methods, which are fulfilled by classes that implement the interface. An interface is often described as a *contract* between the client and the service implementation. The interface defines precisely what the client can expect the service implementation to perform and provide upon completion of the activity.

Our first business service method searches the database for all issues and returns them to the client.

IssueService Interface

```
1 package org.example.issuetracker.service;
2
3 import java.util.List;
4
5 import org.example.issuetracker.model.jpa.Issue;
6
7 public interface IssueService {
8
9     /**
10      * Search the issue data repository for all Issue entities.
11      * @return A List of Issue entities or null if none found.
12      */
13     List<Issue> findAll();
14
15 }
```

The interface defines a single public method, `findAll`. It is always a good idea to include comments that improve the readability of your logic. The `findAll` method is described by a JavaDoc comment briefly describing what it does and what it returns to clients.

Note that the method modifier `public` has been omitted from the method declaration. By default, all methods declared in a Java interface are public. Inside the service implementation class, methods may be declared with reduced access levels such as `protected` or `private`. However, those may not be invoked by service clients and, therefore, are not defined on the service interface.

Creating the Implementation Class

Let's complete the business service by creating the service implementation class.

IssueServiceBean Class

```
1 package org.example.issuetracker.service;
2
3 import java.util.List;
4
5 import org.example.issuetracker.model.jpa.Issue;
6 import org.example.issuetracker.repository.jpa.IssueRepository;
7 import org.slf4j.Logger;
8 import org.slf4j.LoggerFactory;
9 import org.springframework.beans.factory.annotation.Autowired;
10 import org.springframework.stereotype.Service;
11
12 @Service
13 public class IssueServiceBean implements IssueService {
14
15     private Logger logger = LoggerFactory.getLogger(this.getClass());
16
17     @Autowired
18     private IssueRepository issueRepository;
19
20     @Override
21     public List<Issue> findAll() {
22         logger.info("> findAll");
23
24         List<Issue> issues = issueRepository.findAll();
25
26         logger.info("< findAll");
27         return issues;
28     }
29
30 }
```

The `@Service` annotation is a class-level stereotype annotation that declares this class to be a service component. When a Spring application starts, a *component scanner* searches for classes with stereotype annotations. When it discovers a class annotated with `@Service`, the component scanner registers the class making Spring aware of its existence. When our application starts, the component scanner will register `IssueServiceBean` as a business service implementation class and `IssueService` as the client interface for the service.

The `IssueServiceBean` class implements the `IssueService` interface and, therefore, provides an implementation for the `findAll` method. In addition to the method implementation, the `IssueServiceBean` has several other facets that are common to business service implementations

The `logger` is a SLF4J, or Simple Logging Framework for Java, Logger implementation. The SLF4J dependency is provided by the Spring Boot Starter Parent POM, making it unnecessary for it to be declared explicitly in our project POM file. Logger statements are used to trace the application flow, describe the state of the application, and provide detail when exceptions occur. In the `findAll` method, the logger is used to record the entrance into and exit from the method. Should a problem arise and we need to troubleshoot it with the application logs, using statements like these throughout the application provides insight into the application flow.

The `@Autowired` annotation instructs Spring to inject a dependency into an attribute or method. In this case, we are instructing Spring to inject an instance of the Spring Data JPA implementation class for the `IssueRepository` interface. Remember that the `IssueRepository` used the `@Repository` annotation. Similar to the `@Service` annotation, the `@Repository` annotation is a stereotype annotation that makes the class available for dependency injection. The `@Autowired` annotation is way to say *“inject an implementation class of this type here.”*

The injected `issueRepository` is used to query the database for `Issue` entities and conveniently returns them as a `java.util.List`. If the database query does not return any information, the list is `null`.

Excluding the logging statements, the `findAll` method has just one line of code to search the database. Why go to the trouble to create a business service at all? Why not just inject and use the `IssueRepository` in the web service controller class directly? Both of these questions are valid, especially when the `IssueServiceBean.findAll` method contains just one relevant line of code.

In most cases, business service methods have more logic, even in simple finder methods. Business service methods implement logic to validate method parameters, enforce fine-grained security rules, and massage data retrieved from the database before returning it to the client. Whether it is one line of code or many, locating all the business logic for `Issue` entities within the `IssueService` promotes reuse. In a more complex application, the `IssueService` may be injected into multiple client classes.

To achieve high cohesion in the application, we ensure that all business logic concerned with the `Issue` entity is encapsulated within the `IssueServiceBean` because it is the responsibility of the business service.

Creating the Controller

Controller components handle HTTP web requests. They encapsulate request validation, request parsing, invocation of one or more business services, and response formatting. The Spring `@Controller` stereotype annotation identifies controller components and enables them for instrumentation and management by the Spring `ApplicationContext`. In the issue tracker application, we use a specialization, `@RestController`, that includes all the features of `@Controller` and additionally informs Spring that the controller methods are RESTful web service endpoints. In `@RestController` classes, Spring automatically converts response objects or collections into JSON or XML. Similarly, if a web service endpoint receives data in a HTTP request body, Spring automatically converts JSON

or XML into Java objects or collections. The controller methods that serve as web service endpoints are configured through the use of method and parameter annotations.

Let's create the `IssueController` class to serve as the application layer responsible for handling web service requests for the `Issue` entity. First, create a package named `org.example.issuetracker.web.api` within the main application code structure, `src/main/java`. Within the new package, create the `IssueController` class illustrated below.

IssueController Class

```
1 package org.example.issuetracker.web.api;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 import org.example.issuetracker.model.jpa.Issue;
7 import org.example.issuetracker.service.IssueService;
8 import org.slf4j.Logger;
9 import org.slf4j.LoggerFactory;
10 import org.springframework.beans.factory.annotation.Autowired;
11 import org.springframework.http.HttpStatus;
12 import org.springframework.http.MediaType;
13 import org.springframework.http.ResponseEntity;
14 import org.springframework.web.bind.annotation.RequestMapping;
15 import org.springframework.web.bind.annotation.RequestMethod;
16 import org.springframework.web.bind.annotation.RestController;
17
18 @RestController
19 public class IssueController {
20
21     private Logger logger = LoggerFactory.getLogger(this.getClass());
22
23     @Autowired
24     private IssueService issueService;
25
26     @RequestMapping(
27         value = "/issues",
28         method = RequestMethod.GET,
29         produces = MediaType.APPLICATION_JSON_VALUE)
30     public ResponseEntity<List<Issue>> getAllIssues() {
31         logger.info("> getAllIssues");
32
33         List<Issue> issues = null;
34         try {
```

```
35         issues = issueService.findAll();
36
37         // if no issues found, create an empty list
38         if (issues == null) {
39             issues = new ArrayList<Issue>();
40         }
41     } catch (Exception e) {
42         logger.error("Unexpected Exception caught.", e);
43         return new ResponseEntity<List<Issue>>(issues,
44             HttpStatus.INTERNAL_SERVER_ERROR);
45     }
46
47     logger.info("< getAllIssues");
48     return new ResponseEntity<List<Issue>>(issues, HttpStatus.OK);
49 }
50
51 }
```

The Spring Framework implements most of the boilerplate logic for us, leaving only the specific details of the web service to be implemented within the controller class.

@RestController

The `@RestController` annotation is a class-level stereotype that declares this class to be a RESTful web service component. When the Spring application starts, the component scanner registers the class, making Spring aware of its existence. When our application starts, the component scanner will register `IssueController` as a RESTful web service controller class.

When Spring encounters a component annotated with `@RestController`, it examines the class for methods annotated with `@RequestMapping`, discussed in the next section, and records each of those methods as distinct web service endpoints.

@RequestMapping

The `@RequestMapping` annotation is applied to methods of a `@RestController` class to declare that method to be a web service endpoint. Further configuration of methods annotated with `@RequestMapping` is supplied via the `value`, `method`, `accepts`, and `produces` elements.

The `IssueController.getAllIssues` method is annotated with the following:


```
@RequestMapping(  
    value = "/issues",  
    method = RequestMethod.GET,  
    produces = MediaType.APPLICATION_JSON_VALUE)
```

The `value` element contains the HTTP context path to which the web service endpoint is mapped. Since the embedded Apache Tomcat server within Spring Boot listens for HTTP requests on `localhost:8080` by default, this web service endpoint is mapped to `http://localhost:8080/issues`. The `value` element is required.

The `method` element defines the HTTP method to which this web service endpoint is mapped. The `RequestMethod` enum is used to configure a `RequestMapping` for one or more specific HTTP methods. If the `method` element is omitted, the web service endpoint will be mapped to all HTTP methods.

The `consumes` element defines the expected mime type of the data in the HTTP request body. Since the `getAllIssues` web service does not receive a request body, the `consumes` element is not declared within the `@RequestMapping` annotation. When the `consumes` element is present, Spring will only invoke the web service endpoint for HTTP requests containing a `Content-Type` header whose value matches the `consumes` element value. The `consumes` element value is also used by Spring to parse the request body data into Java objects or collections. If the `consumes` element is omitted, the web service endpoint will be invoked for requests containing any value in the `Content-Type` HTTP header.

The `produces` element defines the mime type of the data returned in the HTTP response body from the web service endpoint. When the `produces` element is present, Spring will only invoke the web service endpoint for HTTP requests containing an `Accepts` header whose value matches the `produces` element value. The `produces` element value is also used by Spring to format the response body from Java objects or collections. If the `produces` element is omitted, the web service endpoint will be invoked for requests containing any value in the `Accepts` HTTP Header and responses are formatted as `text/plain`.

ResponseEntity

The `ResponseEntity` class encapsulates all the properties of the HTTP response from a web service method. It stores values for the HTTP headers, body, and status code. When returned from a mapped `@RestController` method, Spring automatically creates the HTTP response from the contents of the `ResponseEntity` object.

`ResponseEntity` is a typed class. In the mapped `IssueController` method `getAllIssues` the return type is `ResponseEntity<List<Issue>>`. The `ResponseEntity` type, in this case `List<Issue>`, specifies the type of object that will be formatted as the HTTP response body.

The `IssueController.getAllIssues` web service endpoint illustrates returning a `ResponseEntity` with an error status code and a success status code. In the code snippet below, a `ResponseEntity` is constructed with two parameters, a collection of issues and a HTTP status code value of 500

indicating that a problem has occurred. The collection of issues is null and, therefore, the response body will not contain any data.

```
return new ResponseEntity<List<Issue>>(issues, HttpStatus.INTERNAL_SERVER_ERROR);
```

In the next snippet, a `ResponseEntity` is constructed with the same two parameters: the collection of issues and a HTTP status code value of 200 indicating a successful execution of the web service. The collection of issues contains zero to many `Issue` objects retrieved from the database. Since the `@RequestMapping` produces value is `application/json`, Spring converts the `List<Issue>` into a JSON array of objects whose attributes mirror those of the Java `Issue` class. If the `List<Issue>` is empty, Spring returns an empty JSON array.

```
return new ResponseEntity<List<Issue>>(issues, HttpStatus.OK);
```

Creating the Cross-Origin Filter

Filters provide the ability to pre-process and post-process HTTP requests. The issue tracker user interface may be hosted using a different domain name or port number than the web services. The AJAX requests emanating from the user interface would have a different *origin* than the base URL of the web services. HTTP requests of this type are called *cross-origin* requests. The `Origin` HTTP header value on the HTTP request contains a different domain name or port number than the web services. Browser security prevents sending cross-origin requests.

To allow *cross-origin* requests between the issue tracker application components, let's add a filter class which returns HTTP headers informing the browser that the web services accept cross-origin requests.

First, create a package named `org.example.issuetracker.web.filter` within the main application code structure, `src/main/java`. Within the new package, create the `SimpleCORSFilter` class illustrated below.

SimpleCORSFilter Class

```
1 package org.example.issuetracker.web.filter;
2
3 import java.io.IOException;
4
5 import javax.servlet.Filter;
6 import javax.servlet.FilterChain;
7 import javax.servlet.FilterConfig;
8 import javax.servlet.ServletException;
9 import javax.servlet.ServletRequest;
10 import javax.servlet.ServletResponse;
```

```
11 import javax.servlet.http.HttpServletResponse;
12
13 import org.springframework.stereotype.Component;
14
15 @Component
16 public class SimpleCORSFilter implements Filter {
17
18     @Override
19     public void destroy() {
20
21     }
22
23     @Override
24     public void doFilter(ServletRequest req, ServletResponse res,
25         FilterChain chain) throws IOException, ServletException {
26         HttpServletResponse response = (HttpServletResponse) res;
27         response.setHeader("Access-Control-Allow-Origin", "*");
28         response.setHeader("Access-Control-Allow-Methods",
29             "DELETE, GET, OPTIONS, PATCH, POST, PUT");
30         response.setHeader("Access-Control-Max-Age", "3600");
31         response.setHeader("Access-Control-Allow-Headers",
32             "x-requested-with, content-type");
33         chain.doFilter(req, res);
34     }
35
36     @Override
37     public void init(FilterConfig filterConfig) throws ServletException {
38
39     }
40
41 }
```

The browser first sends a request using the HTTP OPTIONS method. The `doFilter` method processes inbound HTTP requests returning a set of access control HTTP headers on the HTTP response. These headers inform the browser that the web services accept cross-origin requests. The browser proceeds with the GET, POST, PUT, PATCH, or DELETE AJAX calls.

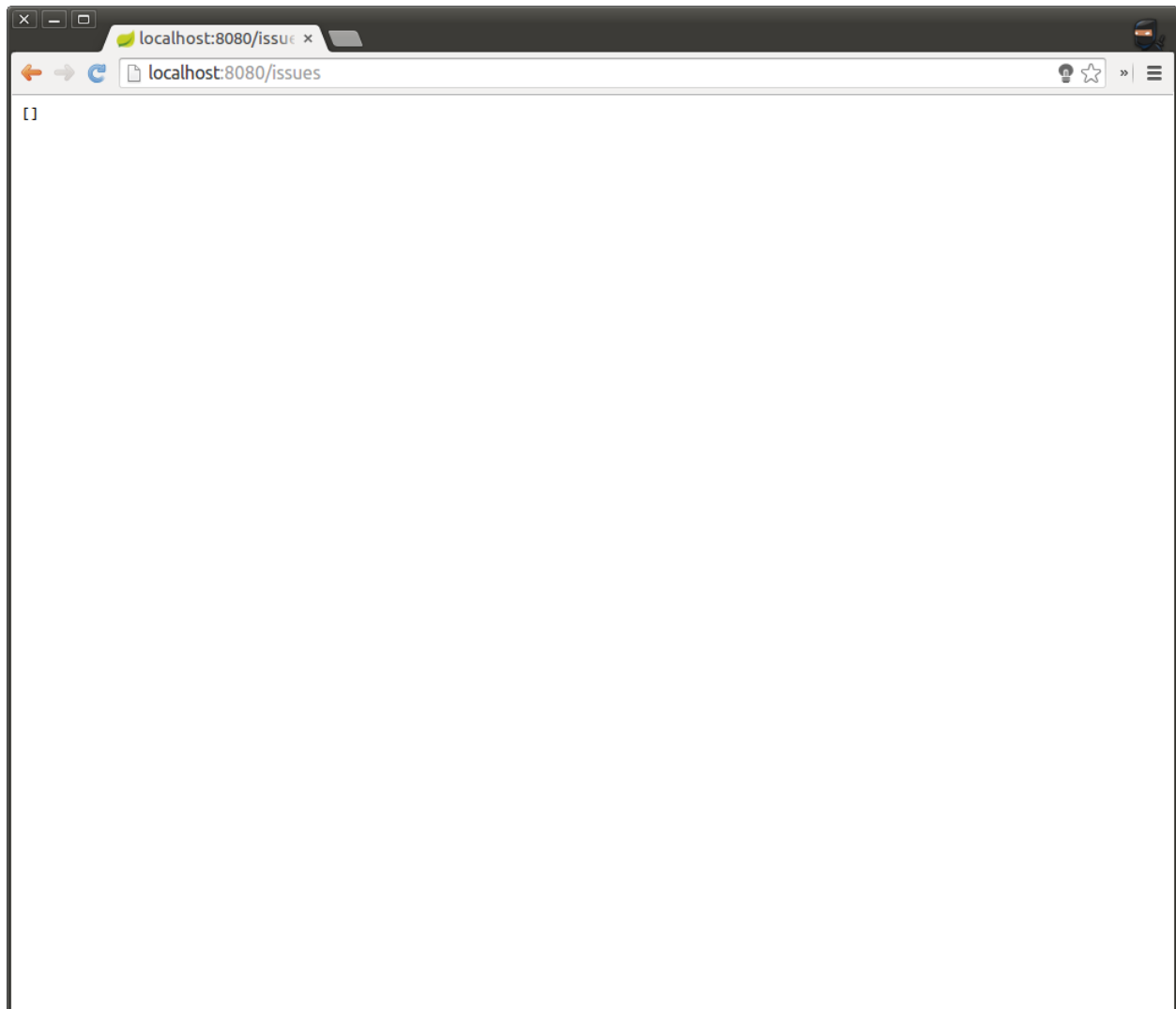
The filter class is annotated with the `@Component` stereotype. When the application starts, Spring recognizes the class as a filter and automatically registers it to handle all inbound HTTP requests.

Running the Application

Congratulations! You have just created the first RESTful web service. Let's start the application again and test the code. Open a terminal window and navigate to the base project directory where the `pom.xml` file is located. Type `mvn spring-boot:run` and press *Enter*. The application generates a few more lines of logging than when we first started the application in the last chapter. That is to be expected. The application is configuring itself for database persistence and a web service endpoint.

When the embedded Apache Tomcat server startup has completed successfully, the console log prints a line similar to `Started Application in 5.483 seconds (JVM running for 5.833)` near the end.

The simplest method to test the `getAllIssues` web service is through the browser. Open your favorite browser. Type `http://localhost:8080/issues` into the address bar and press the *Enter* key.



Get All Issues Browser Test

The web service returns an empty JSON array. We have not created any issues in our database yet, so the request to retrieve all of the issues returns an empty `List<Issue>` collection, which Spring converts to an empty JSON array, `[]`, and returns in the HTTP response body.

Switch to the terminal window where the application is running. In the application console log, the statements depicting the flow of control as the web service processed the request are illustrated below.

Console Log

```
INFO 12508 --- [4] o.e.i.web.api.IssueController      : > getAllIssues
INFO 12508 --- [4] o.e.i.service.IssueServiceBean    : > findAll
INFO 12508 --- [4] o.e.i.service.IssueServiceBean    : < findAll
INFO 12508 --- [4] o.e.i.web.api.IssueController      : < getAllIssues
```



The Road Ahead

In the next few chapters we will bootstrap the user interface application and consume the *Get All Issues* web service. Then, we will incrementally enhance the web services and user interface to create, update, and delete issues.

Part III. User Interface Foundation

Bootstrap the User Interface Project

Installing the Dependencies

The user interface project requires two third-party libraries to be installed on your computer, [node.js](http://nodejs.org/)²⁶ and the Grunt Command Line Interface, better known as [Grunt CLI](http://gruntjs.com/getting-started).²⁷ We will use node.js and Grunt in a manner analogous to how we used Apache Maven for the web services project.

The user interface application is written in the JavaScript programming language. JavaScript is supported by all major web browsers, so you do not need to install any libraries.

Node.js is a platform built atop the Google Chrome web browser's JavaScript runtime environment. Node.js and Grunt perform build, package, and hosting capabilities for the user interface application to support local development activities.

Grunt is a task management and execution plugin for node.js. The Grunt CLI facilitates the execution of Grunt tasks on an application project.

node.js and NPM

Every web browser ships with a built-in JavaScript runtime environment. Node.js is a JavaScript runtime environment outside of a browser. In fact, the foundation of node.js is the Google Chrome JavaScript engine. Organizations use node.js for a wide array of purposes ranging from unit testing to a full-fledged web application server. For this project, node.js is used in conjunction with Grunt to facilitate building, packaging, and hosting the user interface application.

The node package manager, or *npm*, is a utility that manages node.js dependencies. Packages for node.js may be installed globally, one installation on the host machine, or at the project level. This flexibility allows software engineers to use different versions of node.js packages across projects without causing painful conflicts. Later in this chapter, we will author the `package.json` file for the user interface project. The `package.json` file is much like the `pom.xml` file for Apache Maven in that it informs node.js and npm about project dependencies.

Node.js is installed differently depending upon the host operating system. The [node.js downloads](http://nodejs.org/download)²⁸ page is the best place to obtain the installers for Microsoft Windows and Mac OS X. Linux users should follow Joyent's [installing node.js via package manager](https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager)²⁹ wiki page on GitHub. These instructions ensure that the latest version of Node.js is installed on the host machine.

²⁶<http://nodejs.org/>

²⁷<http://gruntjs.com/getting-started>

²⁸<http://nodejs.org/download/>

²⁹<https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager>

After completing the installation, open a terminal window and type `node --version` to verify the installation.

```
$ node --version  
v0.10.32
```

If the node.js version is printed, the installation is successful. If not, retrace your steps through the installation guide.

To verify the node package manager, or NPM, installation type `npm --version`.

```
$ npm --version  
1.4.28
```

Grunt CLI

Grunt is a node.js package that manages task workflow automation. Grunt has a rich library of plugins that provide hundreds of building blocks to fulfill project specific requirements. In the issue tracker user interface project, we will use Grunt to perform tasks such as cleaning our project workspace, copying files to a distribution directory, packaging a distributable application, and running a web server for local development.

Grunt workflow tasks are executed in a terminal window using the Grunt Command Line Interface, or CLI. The Grunt CLI package is a node.js package and is installed using the node package manager utility, npm. Open a terminal window and execute the following command to install the Grunt CLI.

```
$ npm install -g grunt-cli
```

The command above instructs the node package manager to download and install the `grunt-cli` node.js package on the host machine. The `-g` option instructs the package manager to make the `grunt-cli` executables globally available on the host machine. To verify the installation of the Grunt CLI package, type `grunt --version` at a terminal prompt and press *Enter*.

```
$ grunt --version  
grunt-cli v0.1.13
```

If the Grunt CLI version is printed, the installation is successful. If not, retrace your steps through the installation guide.

Choosing a Source Editor

There are several popular editors available for editing the type of source code files in the user interface project. For the purposes of this book, you may use your favorite text editor or the default text editor that is shipped with your operating system. You may also import this project into the Spring Tool Suite IDE as a general project and use the built in JavaScript and CSS editors.

Structuring the Project

Let's begin building the user interface project. Create a directory structure on your local machine as depicted below.

```
issue-tracker-ui
├── lib
├── src
│   ├── main
│   │   └── app
│   │       ├── css
│   │       ├── js
│   │       │   └── modules
│   └── templates
```

The `issue-tracker-ui` directory is the base directory. In the next sections, we will create files in the base directory that are used by `node.js` and `Grunt`.

Immediately beneath the base directory, create a sub-directory named `lib`. The user interface project has a few third-party CSS and JavaScript dependencies like `Bootstrap` and `jQuery`. Those dependencies will be placed in the `lib` directory.

Also beneath the base directory, create a sub-directory named `src`. The `src` directory is the parent directory containing all of the project's application source code. Continue to create the `src` directory structure as depicted above. The directory structure will be discussed in greater detail later in the book as the application components are constructed.



Unit Test Structure

Unit tests are an invaluable part of any software project. To add unit test modules to this project, a `test/app/js` structure would be created below the `src` directory.

The lib Directory

The issue tracker user interface has a few open source, third party dependencies used as a foundation for the application style and behavior. Let's examine those dependencies and briefly introduce each of them.

Backbone.Marionette

The Backbone.Marionette, a.k.a. [Marionette](http://marionettejs.com/)³⁰, JavaScript library extends the popular Backbone library and provides additional features to simplify client-side user interface application development. One of the main goals of the Marionette library is to reduce or eliminate boilerplate code required in pure Backbone applications, thus streamlining the development process by reducing number of lines of code to be written. The Marionette library does more than just extend Backbone classes though. It contains additional JavaScript classes that realize the most frequently employed design patterns for Backbone-based applications.

Backbone

[Backbone](http://backbonejs.org/)³¹ is one of the most popular open source JavaScript Model-View-Presenter libraries available today. Backbone's peers include Angular, Ember, Knockout, and others. There are many opinions regarding the differences between these libraries, but one of the major differences is that the Backbone library provides a great amount of implementation flexibility. Some may argue that this flexibility means that it requires that more logic be implemented by the engineer. This is true to an extent; however, flexibility is beneficial when teams are faced with a myriad of requirements and constraints which are out of their control.

Underscore

The [Underscore](http://underscorejs.org/)³² JavaScript library is a dependency of the Backbone and Backbone.Marionette libraries. Underscore provides a suite of utility functions for operations on collections, arrays, objects, events and more. Additionally, the Underscore library has a built-in HTML template language and compiler. HTML templates are snippets of HTML with behavioral logic, such as conditional statements and loops, intermixed. Backbone and Marionette use the Underscore library behind the scenes to dynamically render HTML using templates and a JSON data model.

jQuery

The [jQuery](http://jquery.com/)³³ JavaScript library is also a dependency of Backbone. jQuery offers HTML DOM traversal, selection, and manipulation. The jQuery library is used by Backbone to make AJAX calls to web services to retrieve and modify the data managed by the user interface.

³⁰<http://marionettejs.com/>

³¹<http://backbonejs.org/>

³²<http://underscorejs.org/>

³³<http://jquery.com/>

log4javascript

The [log4javascript](http://log4javascript.org/)³⁴ library is a logging framework for JavaScript applications. It is similar conceptually to the Log4J Java logging framework. Log4javascript is a cross-browser solution, meaning that developers do not need to address traditional console logging in Internet Explorer.

Bootstrap

The [Bootstrap](http://getbootstrap.com/)³⁵ framework provides a foundation of HTML, CSS, and JavaScript to structure and style web application projects. Bootstrap features a 12-column grid system to structure HTML content that is responsive out-of-the-box, meaning that your web application will seamlessly adjust to display on mobile devices. Using just HTML and CSS, Bootstrap provides a beautiful user experience for navigation, forms, tables, images, and much more. By including Bootstrap's JavaScript in the project, engineers may also create modal dialogs, tooltips, image carousels, accordion boxes, and more.

Font Awesome

The [Font Awesome](http://fontawesome.io/)³⁶ framework is a CSS iconography framework that compliments Bootstrap. Use Font Awesome icons in Bootstrap navigation links, buttons, or anywhere else in HTML. Font Awesome icons are created as scalable vector graphics and delivered as a font library. Therefore, any of Bootstrap's font colors may be used to style Font Awesome icons. All of the icons can be layered, rotated, and inverted to create a variety of effects. The Font Awesome framework also contains a set of spinner icons which are useful indications to the user that the application is hard at work.

Downloading the Libraries

Download [Bootstrap](http://getbootstrap.com/)³⁷, [Font Awesome](http://fontawesome.io/)³⁸, [jQuery](http://jquery.com/download/)³⁹, [Underscore](http://underscorejs.org/)⁴⁰, [Backbone](http://backbonejs.org/)⁴¹, [log4javascript](http://log4javascript.org/)⁴², and [Marionette](http://marionettejs.com/#download)⁴³. See Appendix A for the dependency versions used by this book.

Open the Bootstrap archive that you downloaded and save the entire contents to the project `lib` directory. Rename the top-level directory of the unpacked Bootstrap archive to be `bootstrap-M-m-p` where the M-m-p suffix represents the Bootstrap version number.

³⁴<http://log4javascript.org/>

³⁵<http://getbootstrap.com/>

³⁶<http://fontawesome.github.io/Font-Awesome/>

³⁷<http://getbootstrap.com/>

³⁸<http://fontawesome.github.io/Font-Awesome/>

³⁹<http://jquery.com/download/>

⁴⁰<http://underscorejs.org/>

⁴¹<http://backbonejs.org/>

⁴²<http://sourceforge.net/projects/log4javascript/>

⁴³<http://marionettejs.com/#download>

```
lib
└─ bootstrap-3.3.5
   ├── css
   │   ├── bootstrap.css
   │   ├── bootstrap.css.map
   │   ├── bootstrap.min.css
   │   ├── bootstrap-theme.css
   │   ├── bootstrap-theme.css.map
   │   └─ bootstrap-theme.min.css
   ├── fonts
   │   ├── glyphsicons-halflings-regular.eot
   │   ├── glyphsicons-halflings-regular.svg
   │   ├── glyphsicons-halflings-regular.ttf
   │   ├── glyphsicons-halflings-regular.woff
   │   └─ glyphsicons-halflings-regular.woff2
   └─ js
       ├── bootstrap.js
       └─ bootstrap.min.js
```

Open the Font Awesome archive and repeat the steps performed with the Bootstrap archive. Rename the top-level directory of the unpacked Font Awesome archive to be fontawesome-M-m-p. You may optionally choose to remove the fontawesome/less and fontawesome/scss directories from the unpacked contents.

```
lib
└─ fontawesome-4.3.0
   ├── css
   │   ├── font-awesome.css
   │   └─ font-awesome.min.css
   └─ fonts
       ├── FontAwesome.otf
       ├── fontawesome-webfont.eot
       ├── fontawesome-webfont.svg
       ├── fontawesome-webfont.ttf
       └─ fontawesome-webfont.woff
```

Open the log4javascript archive and save the file named log4javascript.js to the project lib directory as log4javascript-M-m-p.js. Again, the -M-m-p suffix to the filename should be the library version such as log4javascript-1.4.13.js. Repeat this for jQuery, Underscore, Backbone, and Marionette.

From the Backbone.Marionette archive also save the file named json2.js to the project lib directory.

The complete contents of the lib directory are depicted below.

```

lib
├─ backbone-1.2.1.min.js
├─ backbone.marionette-2.4.2.min.js
├─ backbone.marionette.map
├─ backbone-min.map
├─ bootstrap-3.3.5
│   └─ css
│       ├── bootstrap.css
│       ├── bootstrap.css.map
│       ├── bootstrap.min.css
│       ├── bootstrap-theme.css
│       ├── bootstrap-theme.css.map
│       └─ bootstrap-theme.min.css
│   └─ fonts
│       ├── glyphsicons-halflings-regular.eot
│       ├── glyphsicons-halflings-regular.svg
│       ├── glyphsicons-halflings-regular.ttf
│       ├── glyphsicons-halflings-regular.woff
│       └─ glyphsicons-halflings-regular.woff2
│   └─ js
│       ├── bootstrap.js
│       └─ bootstrap.min.js
├─ fontawesome-4.3.0
│   ├── css
│   │   ├── font-awesome.css
│   │   └─ font-awesome.min.css
│   └─ fonts
│       ├── FontAwesome.otf
│       ├── fontawesome-webfont.eot
│       ├── fontawesome-webfont.svg
│       ├── fontawesome-webfont.ttf
│       └─ fontawesome-webfont.woff
├─ jquery-2.1.4.min.js
├─ jquery-2.1.4.min.map
├─ json2.js
├─ log4javascript-1.4.13.js
├─ underscore-1.8.3.min.js
└─ underscore-min.map

```

The src Directory

The src directory structure is the location of the application source files for the issue tracker user interface. The src/main structure contains the core application source files. It will not be discussed

in this book; however, the `src/test` directory is where unit test source files would be located.

app.css

Let's create a source file to contain the application CSS. The Bootstrap and Font Awesome frameworks perform most of the styling for us, but an application CSS file is required for some minor adjustments.

In the `src/main/app/css` directory, create a file named `app.css`.

app.css

```
1  /* Project Styles */
2  /* Use bootstrap styles first and override only if necessary. */
```

At this time the file is nothing more than a placeholder. Later in the book, styles are added to enhance the base Bootstrap user experience.

index.html

In a single page application, or SPA, the `index.html` page is the *only* HTML page in the entire application, hence the name *single* page application. The page contains links to the third-party dependencies and defines the high-level HTML structural layout of the user interface.

In the `src/main/app` directory, create a file named `index.html`. The contents of the page are shown below.

index.html

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4  <meta charset="utf-8">
5  <meta http-equiv="X-UA-Compatible" content="IE=edge">
6  <meta name="viewport" content="width=device-width, initial-scale=1">
7  <title>Issue Tracker</title>
8  <link href="assets/lib/bootstrap-3.3.5/css/bootstrap.min.css" rel="stylesheet">
9  <link href="assets/lib/fontawesome-4.3.0/css/font-awesome.min.css" rel="styleshe\
10 et">
11 <link href="assets/app/css/app.css" rel="stylesheet">
12 </head>
13 <body>
14   <header id="header-region"> </header>
15
```

```
16 <section id="main-region"> </section>
17
18 <section id="dialog-region"> </section>
19
20 <footer id="footer-region"> </footer>
21
22 <script src="assets/lib/log4javascript-1.4.13.js"></script>
23 <script src="assets/lib/jquery-2.1.4.min.js"></script>
24 <script src="assets/lib/json2.js"></script>
25 <script src="assets/lib/underscore-1.8.3.min.js"></script>
26 <script src="assets/lib/backbone-1.2.1.min.js"></script>
27 <script src="assets/lib/backbone.marionette-2.4.2.min.js"></script>
28 <script src="assets/lib/bootstrap-3.3.5/js/bootstrap.min.js"></script>
29 <script src="assets/app/js/app-templates-1.0.0.js"></script>
30 <script src="assets/app/js/app-1.0.0.min.js"></script>
31 </body>
32 </html>
```

The page is patterned after the [Bootstrap basic page template](http://getbootstrap.com/getting-started/#template)⁴⁴ for HTML5 pages. Note the use of the HTML5 document type, `html`, and `meta charset` tags and attribute values.

Additional `link` tags have been added to load both the Font Awesome and our application CSS files.

The body of the page contains tags that define the high-level layout of the application. Using the `header`, `section`, and `footer` HTML5 tags, the page is structured in the classic three section document format with a twist. We will use the `section id="dialog-region"` area of the page as a common location for displaying modal dialog boxes.

Finally, a series of `script` tags loads the third party and application JavaScript assets. The `script` tags are located at the bottom of the page body so that browsers retrieve them from the server last. This does not have an impact on the issue tracker application because the page does not contain any HTML tags that immediately display content to the user such as logo images. However, it is a best practice to locate `script` tags at the end of the document body so that the browser may immediately render visible HTML page markup prior to downloading the `script` source files.

package.json

The `package.json` file was briefly discussed when installing `node.js`. Let's explore this file in a bit more detail and learn how it will be used in the project.

In a traditional `node.js` package project, the `package.json` file contains all of the metadata describing the project. A `node.js` package project is typically published to a repository and made accessible to other projects by the node package manager. When the `node.js` package is published to a node

⁴⁴<http://getbootstrap.com/getting-started/#template>

package manager repository, the contents of the `package.json` file are used to index, classify, and store the package contents.

In a single page application project, the `package.json` file is used to manage Grunt and the Grunt plugins required to build, package, and locally host the application. The project will not be published to a node.js repository and, therefore, we use only those attributes of the `package.json` file necessary for node.js dependency management.

Let's create the `package.json` file for the issue tracker user interface application. In the project base directory, create the `package.json` file. The contents of the file are shown below.

`package.json`

```
1 {
2   "name": "issue-tracker-ui",
3   "description": "Issue Tracker user interface.",
4   "version": "1.0.0",
5   "dependencies": {},
6   "devDependencies": {
7     "grunt": "^0.4.5",
8     "grunt-contrib-uglify": "^0.6.0",
9     "grunt-contrib-connect": "^0.8.0",
10    "grunt-contrib-jst": "^0.6.0",
11    "grunt-contrib-jshint": "^0.10.0",
12    "grunt-contrib-copy": "^0.7.0",
13    "grunt-contrib-clean": "^0.6.0",
14    "grunt-contrib-watch": "^0.6.1",
15    "grunt-contrib-compress": "^0.13.0"
16  }
17 }
```

The name and version attributes are the only required attributes in a `package.json` file. The description attribute may be included to provide concise details about the project. The dependencies and devDependencies attributes accept a JSON object hash containing dependencies and their versions. The dependencies attribute specifies project runtime dependencies. The devDependencies attribute specifies libraries required to modify the project source code. Since we are using node.js as a development tool, the project dependencies are declared in the devDependencies section.

Gruntfile.js

Grunt is a node.js plugin that provides workflow definition and execution. The workflow configuration is defined in a file named `Gruntfile.js` which is commonly called the *Gruntfile*.

Grunt has a rich, community driven plugin library. Each plugin typically performs one operation such as copying, minifying, or deleting files. Each plugin is configured as a *task* in the Gruntfile.

A Gruntfile is comprised of statements that load the plugins, configure the plugin tasks, and register custom tasks. Custom tasks are a named set of plugin tasks. For example, a custom task to create a distribution of minified JavaScript might include the `clean`, `copy`, and `uglify` Grunt plugin tasks.

Let's create the `Gruntfile.js` file for the issue tracker user interface application. In the project base directory, create a file named `Gruntfile.js`. The contents of the file are shown below.

Gruntfile.js

```
1 module.exports = function(grunt) {
2
3   // Grunt Task Configuration
4   grunt.initConfig({
5     // Load the package.json file for reference
6     // Reference package.json attributes here with <%= pkg.attributeName %>
7     pkg : grunt.file.readJSON('package.json'),
8
9     // Define the 'clean' task
10    // Use clean to remove old builds
11    clean : {
12      src : [ 'dist/*' ]
13    },
14
15    // Define the 'compress' task
16    // Use compress to create a distributable tar.gz package
17    compress: {
18      main: {
19        options: {
20          archive: 'dist/issuetracker.tar.gz'
21        },
22        files: [ {
23          expand: true,
24          cwd: 'dist/',
25          src: ['**']
26        } ]
27      }
28    },
29
30    // Define the 'connect' server task
31    // Use the connect server for local development
32    // Go to http://localhost:9000/
33    connect : {
34      server : {
35        options : {
```

```
36         base : 'dist/',
37         port : 9000
38     }
39 }
40 },
41
42 // Define the 'copy' task
43 // Use copy to copy source files to the distribution directory
44 copy : {
45     main : {
46         files : [ {
47             expand : true,
48             cwd : 'src/main/app/',
49             src : [ '*' ],
50             dest : 'dist/',
51             filter : 'isFile'
52         }, {
53             expand : true,
54             cwd : 'src/main/',
55             src : [ 'app/**',
56                 '!app/**/*.html',
57                 '!app/js/**',
58                 '!app/templates/**'
59             ],
60             dest : 'dist/assets/'
61         }, {
62             expand : true,
63             src : [ 'lib/**' ],
64             dest : 'dist/assets/'
65         } ]
66     }
67 },
68
69 // Define the 'jshint' checker task
70 // Use jshint to check JavaScript source files for errors
71 jshint : {
72     gruntfile : [ 'Gruntfile.js' ],
73     main : [ 'src/main/app/**/*.js' ]
74 },
75
76 // Define the 'jst' task
77 // Use JST to compile HTML templates into JavaScript
```

```
78     jst : {
79       compile : {
80         options : {
81           namespace : 'IssueTrackerTemplates',
82           processName : function(filePath) {
83             return filePath.replace(/^src\/main\/app\/templates\/, '')
84               .replace(/\.html$/, '');
85           }
86         },
87         files : {
88           'dist/assets/app/js/app-templates-<%= pkg.version %>.js':
89             'src/main/app/templates/*.html'
90         }
91       }
92     },
93
94     // Define the 'uglify' task
95     // Use uglify to concatenate and minify JavaScript
96     uglify : {
97       options : {
98         banner :
99           '/*! <%= pkg.name %> <%= grunt.template.today("yyyy-mm-dd") %> */\n',
100       sourceMap : true,
101       sourceMapName : 'dist/assets/app/js/app-<%= pkg.version %>.map'
102     },
103     main : {
104       files : {
105         'dist/assets/app/js/app-<%= pkg.version %>.min.js':
106           ['src/main/app/Application.js', 'src/main/app/js/**/*.js']
107       }
108     }
109   },
110
111   // Define the 'watch' task
112   // Watch source files and perform tasks when changes occur
113   watch : {
114     src : {
115       files : [ 'src/**/*.js', 'lib/**/*.js' ],
116       tasks : [ 'dist' ]
117     }
118   }
119 }
```

```
120 });
121
122 // Load the Grunt plugin tasks
123 // Note: Must also exist in package.json
124 grunt.loadNpmTasks('grunt-contrib-clean');
125 grunt.loadNpmTasks('grunt-contrib-compress');
126 grunt.loadNpmTasks('grunt-contrib-connect');
127 grunt.loadNpmTasks('grunt-contrib-copy');
128 grunt.loadNpmTasks('grunt-contrib-jshint');
129 grunt.loadNpmTasks('grunt-contrib-jst');
130 grunt.loadNpmTasks('grunt-contrib-uglify');
131 grunt.loadNpmTasks('grunt-contrib-watch');
132
133 // Register Grunt custom tasks
134 // Run with Grunt CLI with 'grunt' or 'grunt taskName'
135 grunt.registerTask('default', [ 'jshint', 'clean', 'copy', 'jst', 'uglify' ]);
136 grunt.registerTask('dist', [ 'default', 'compress' ]);
137 grunt.registerTask('run', [ 'default', 'connect:server', 'watch' ]);
138
139 };
```

For those readers that have worked with node.js modules, the `module.exports` statement will be familiar. If this looks strange to you, don't worry. This syntax is specific to node.js module definitions and we will only use it within the Gruntfile.

Within the `initConfig` function, each of the Grunt tasks are configured. The project uses the `clean`, `copy`, `jst`, `uglify`, and `compress` plugins to build and package the application distribution. The `clean` task deletes files and directories. The `copy` task copies files and directories. The `jst` task converts HTML templates into JavaScript functions, concatenating them into a single distribution file. The `uglify` task concatenates all of the JavaScript source files into a single, minified distribution file. The `compress` task creates a single application artifact for server deployment.

The `jshint` task runs a JavaScript lint source code error checker against the project and halts the build process when errors are found. The `jshint` task can locate syntax errors, but cannot detect potential runtime issues.

The `connect` task starts a simple HTTP server to host the application for local development testing. The `watch` task polls the source files and, when a change is detected, rebuilds the application distribution automatically.

A The Road Ahead

In the next chapter, we will begin to construct the JavaScript application while learning how to promote high cohesion using Marionette modules.

Getting Started with Marionette

The popularity of pure client-side user interfaces has exploded in the last decade. The term Single Page Application was coined almost a decade ago and at that time, software engineers began creating foundation classes and frameworks in JavaScript. In the years that followed, open-source JavaScript application frameworks began to emerge. One of those frameworks is Backbone.js, or just Backbone. The Backbone library was first released in 2010 and provided the flexibility, structure and design patterns that engineers had been seeking. As engineers began creating applications with Backbone, it became apparent to some that the gift of flexibility came at a small cost. That cost came in the form of boilerplate code that was often repeated throughout an application. Enter the Backbone.Marionette.js library, or just Marionette. Marionette was created by Backbone aficionados seeking to extend the power of the Backbone library while significantly reducing the need to author boilerplate code. Marionette achieves that goal while simultaneously expanding the number of patterns available to software engineers.

Let's start building the issue tracker user interface with Marionette.

The Marionette Application

In the `src/main/app/js` directory, create a new file named `Application.js`. Similar to the `Application.java` class that we created for the web services, the `Application.js` file contains the source code that defines the main application and its runtime entry point.

Declaring the Application

First declare a variable for the issue tracker application and create a new Marionette Application instance.

Declaring the Application

```
1 // Create the Application
2 window.IssueTrackerApp = new Backbone.Marionette.Application();
```

The variable named `IssueTrackerApp` is attached to the browser Window object. When JavaScript is executed within a web browser, all global variables are attached to the browser's Window object. If care is not taken, name collisions result in variable values being unintentionally overwritten. If the application had been named simply `app`, it increases the likelihood that other JavaScript executing in the browser may overwrite such a generic variable name. To reduce the chance of this occurring,

the application variable is named `IssueTrackerApp`. With a few exceptions, the complete application will be contained within the `IssueTrackerApp` rather than attaching a multitude of objects to the browser `Window` object.

The Marionette Application class manages the lifecycle of all the components. The Application class also serves as the communication hub for application components, leveraging the [Backbone.Wreqr](https://github.com/marionettejs/backbone.wreqr)⁴⁵ library. The next chapter provides an in-depth discussion of Marionette messaging.

Adding Regions

The Application also manages the top-level regions that are defined on the `index.html` page. Through the Application object, regions can be added, removed, shown, and hidden. In the body of the HTML page illustrated below, we created tags to serve as containers for four application regions.

```
<header id="header-region"> </header>

<section id="main-region"> </section>

<section id="dialog-region"> </section>

<footer id="footer-region"> </footer>
```

In the `Application.js` file, let's add the four regions to the `IssueTrackerApp`. The `addRegions` function accepts a parameter containing a hash of region names mapped to their CSS selector. Each region has an `id` attribute which is used as the selector to pair the HTML DOM element to a named Marionette Region.

Create the Regions

```
1 // Create the top-level Regions
2 IssueTrackerApp.addRegions({
3   headerRegion : '#header-region',
4   mainRegion   : '#main-region',
5   dialogRegion : '#dialog-region',
6   footerRegion : '#footer-region'
7 });
```

The application object, `IssueTrackerApp`, exposes a set of functions to manage each Region. Like the `addRegions` function illustrated above, the application has functions to programmatically remove regions, `removeRegion`, and retrieve regions, `getRegion`. While a region may be retrieved using

⁴⁵<https://github.com/marionettejs/backbone.wreqr>

getRegion, each region is also exposed as a public variable on the application object. For example, the statements `IssueTrackerApp.getRegion('headerRegion')` and `IssueTrackerApp.headerRegion` are identical. The latter form is most commonly used by engineers and is the form used throughout the book to access regions.

Application Events

The Application object publishes events to indicate key lifecycle state changes. We need to perform activities when the Marionette application starts. Let's create an event listener for the *start* event.

Application Start Event Listener

```
1 // Initialize the Router
2 IssueTrackerApp.on('start', function(options) {
3   logger.debug("Backbone.history.start");
4   Backbone.history.start();
5 });
```

The `on` function accepts the name of the event as the first parameter and the callback function to be executed when the event is received as the second parameter. In the code snippet above, an event listener waits for the application to publish an event named *start*. When that event is published, the listener callback function is called. The function starts the `Backbone.history` object.



Backbone.history

The `Backbone.history` object records changes to the application URL in the browser address bar. The `Backbone.history` object is started after the Marionette Application, but before the application performs navigation that alters the URL.

Starting the Application

Now that the application is declared and configured, it must be started. The declaration of `IssueTrackerApp` only instantiated a Marionette Application object and stored it in the browser Window as a named variable. To ensure that the entire HTML document is loaded before starting the application, use jQuery to listen for the publication of the *ready* event indicating that the entire HTML document as well as all externally linked resources have been loaded.

Start the Application

```
1 // Start the Application
2 $( function() {
3   // Configure log4javascript Library
4   window.logger = log4javascript.getLogger();
5   consoleAppender = new log4javascript.BrowserConsoleAppender();
6   consoleAppender.setLayout(new log4javascript.PatternLayout('%d{HH:mm:ss} %-5p \
7 - %m'));
8   window.logger.addAppender(consoleAppender);
9
10  // Start Marionette Application
11  logger.debug("IssueTrackerApp.start");
12  IssueTrackerApp.start();
13 });
```

In the snippet above, the shorthand notation for jQuery's document onReady listener is used, `$(function() { // do stuff })`.

While most of the application is contained within the `window.IssueTrackerApp` object, the `log4javascript` library's `Logger` object must be declared as a global variable to function properly. Thus, it too must be attached to the window. The `Logger` variable is named `window.logger` allowing it to be used throughout the application with statements such as `logger.debug("my message")`. The logger also exposes `trace`, `info`, `warn`, `error`, and `fatal` in addition to the `debug` function.

Finally, the Marionette application is started using the function `IssueTrackerApp.start()`. The application publishes lifecycle startup events, triggering the execution of the `start` callback function.

The complete `Application.js` source file is depicted below.

Application.js

```
1 // Create the Marionette Application
2 window.IssueTrackerApp = new Backbone.Marionette.Application();
3
4 // Create the top-level Regions
5 IssueTrackerApp.addRegions({
6   headerRegion : '#header-region',
7   mainRegion   : '#main-region',
8   dialogRegion : '#dialog-region',
9   footerRegion : '#footer-region'
10 });
11
12 // Initialize the Router
```

```
13 IssueTrackerApp.on('start', function(options) {
14   logger.debug("Backbone.history.start");
15   Backbone.history.start();
16 });
17
18 // On Document Ready Event, Start the Application
19 $( function() {
20   // Configure log4javascript Library
21   window.logger = log4javascript.getLogger();
22   consoleAppender = new log4javascript.BrowserConsoleAppender();
23   consoleAppender
24     .setLayout(new log4javascript.PatternLayout('%d{HH:mm:ss} %-5p - %m'));
25   window.logger.addAppender(consoleAppender);
26
27   // Start Marionette Application
28   logger.debug("IssueTrackerApp.start");
29   IssueTrackerApp.start();
30 });
```

Running the Application

In the last chapter, we created a Grunt task named `run` that packages the application distribution and starts a local web server. Open a terminal window and navigate to the project base directory where the `Gruntfile.js` file is located. Type `npm install` to download the development dependencies declared in the `package.json` file. Next, type `grunt run` at the prompt to start the web server. The terminal output should be similar to that which is shown below.

Start the Web Server

```
$ grunt run
```

```
Running "jshint:gruntfile" (jshint) task
>> 1 file lint free.
```

```
Running "jshint:main" (jshint) task
>> 1 file lint free.
```

```
Running "clean:src" (clean) task
>> 0 paths cleaned.
```

```
Running "copy:main" (copy) task
Created 12 directories, copied 58 files
```

Running "jst:compile" (jst) task

```
>> Destination not written because compiled files were empty.
```

Running "uglify:main" (uglify) task

```
>> 1 sourcemap created.
```

```
>> 1 file created.
```

Running "connect:server" (connect) task

```
Started connect web server on http://0.0.0.0:9000
```

Running "watch" task

```
Waiting...
```



Stopping the Web Server

Press `Ctrl-C` to stop the local web server.

Near the end of the console output, the logging indicates that the web server is started and is listening on port 9000. Open a browser and press the `F12` key to open the developer tools. Go to the console tab to view the output from the application logging statements. In the address bar, type `http://localhost:9000/index.html` and press *Enter*.

On the browser page nothing happens. The page is blank because we have not yet created and shown content in the Marionette application regions. However, the browser console log statements depict the application activity. Look at the console tab in the browser's developer tools.

Console Log

- 1 11:26:58 DEBUG - IssueTrackerApp.start
 - 2 11:26:58 DEBUG - Backbone.history.start
-

Note the order of the log statements. First the `IssueTrackerApp.start` statement is logged. This statement is issued from the document *ready* event listener. The `Backbone.history.start` statement is logged from the Marionette *start* event listener. The Marionette application is ready to show content.

Marionette Modules

Previously, we discussed the architectural property of [cohesion](#). In this section, we organize the Marionette application into modules to promote high cohesion. Leveraging Marionette's [Module](#)⁴⁶ class, we divide the issue tracker user interface into functionally cohesive parts.

⁴⁶<http://marionettejs.com/docs/marionette.module.html>

Marionette modules are commonly composed from a handful of classes such as controllers, routers, views, entity models, and templates. Each of these module components has specific responsibilities to achieve the overall module functionality, further promoting cohesion by separating the responsibilities within a module into classes of a specific type.

The entity models describe the data objects for which the module is responsible. Models may contain behaviors that manipulate their data such as filtering the contents of a collection or formatting an attribute.

The templates contain the static markup and dynamic behavior for generating the user interface views managed by the module. Templates are rendered by the module using entity models to produce dynamic, data-driven HTML markup.

The views are responsible for managing user interaction with the rendered HTML templates. Typically, there is one view class for each template. A view listens for user interactions, in the form of HTML DOM events, that are published within the DOM generated by the view's template. The view handles these events and executes behavior when they occur.

The controllers orchestrate behavior across the classes within a module and manage external communication with other application modules. Some controller responsibilities include interaction with web services to populate entity models; creating and showing views; and publishing messages when events occur which may concern other application modules.

Routers and controllers work together. A router is responsible for mapping the browser's address bar hash value to a controller method. When the value following the # hash symbol in the browser address bar changes, a router invokes the controller function mapped to the matching URL hash.

Displaying Static Content

Let's ease into Marionette modules by creating two modules that display static content, the header and footer. Modules that display static content do not require all of the components described in the previous section. Entity models are not needed because the HTML markup rendered by the views is *static*. The modules do not require routers because they are shown programmatically rather than triggered by the URL in the web browser.

There are benefits to displaying static content in a module rather than just placing the HTML directly in the page. The HTML views rendered by modules may be shown or hidden through application logic. A view can render different HTML templates based upon current conditions. Building a module to render static content does not require many lines of code, but provides the flexibility to quickly convert the module to render dynamic content at a later time.

The Header Module

Let's create a module to display a Bootstrap [navbar](http://getbootstrap.com/components/#navbar)⁴⁷ in the header region of the IssueTrackerApp. In a more sophisticated application, the navbar may contain dynamic navigation links, but in the

⁴⁷<http://getbootstrap.com/components/#navbar>

issue tracker application the navigation is static. The header module consists of an HTML template, a view, and a controller.

Navbar Template

In the `src/main/app/templates` directory, create a new file named `navbar.html`. The file extension `html` is not mandatory, but is often recognized by text editors providing syntax highlighting and code completion features.

Within the `navbar.html` file, add the HTML to create a Bootstrap navbar that is fixed to the top of the page.

`navbar.html`

```

1 <nav class="navbar navbar-default navbar-fixed-top" role="navigation">
2   <div class="container-fluid">
3     <!-- Brand and toggle get grouped for better mobile display -->
4     <div class="navbar-header">
5       <button type="button" class="navbar-toggle collapsed" data-toggle="collaps\
6 e" data-target="#bs-example-navbar-collapse-1">
7       <span class="sr-only">Toggle navigation</span>
8       <span class="icon-bar"></span>
9       <span class="icon-bar"></span>
10      <span class="icon-bar"></span>
11    </button>
12    <a class="navbar-brand" href="#">Issue Tracker</a>
13  </div>
14
15  <!-- Collect the nav links, forms, and other content for toggling -->
16  <div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">
17    <ul class="nav navbar-nav">
18      <li><a href="#">Home</a></li>
19    </ul>
20  </div><!-- /.navbar-collapse -->
21 </div><!-- /.container-fluid -->
22 </nav>

```

Note that this is not a complete HTML page. A template contains the markup for a specific portion of the page.

The application CSS file must be updated to include the spacing prescribed by the [fixed to top](http://getbootstrap.com/components/#navbar-fixed-top)⁴⁸ navbar documentation. Let's add 70 pixels of padding to the top of the page body so that other content on the page is not hidden beneath the navbar.

⁴⁸<http://getbootstrap.com/components/#navbar-fixed-top>

app.css

```
1  /* Project Styles */
2  /* Use bootstrap styles first and override only if necessary. */
3
4  /* body padding for http://getbootstrap.com/components/#navbar-fixed-top */
5  body {
6      padding-top: 70px;
7  }
```

Precompiling Templates

The Grunt file, [Gruntfile.js](#), contains a task named `jst` which utilizes the JST plugin to compile HTML templates into JavaScript functions. The JST task from the project Gruntfile is shown in the excerpt below.

JST Grunt Task

```
1  // Define the 'jst' task
2  // Use JST to compile HTML templates into JavaScript
3  jst : {
4      compile : {
5          options : {
6              namespace : 'IssueTrackerTemplates',
7              processName : function(filePath) {
8                  return filePath.replace(/^src\/main\/app\/templates\//, '')
9                      .replace(/\.html$/, '');
10             }
11         },
12         files : {
13             'dist/assets/app/js/app-templates-<%= pkg.version %>.js':
14                 'src/main/app/templates/*.html'
15         }
16     }
17 }
```

The `files` section of the task configuration instructs JST to compile templates found in the `src/main/app/templates` directory with the file extension `html`. The compiled templates are concatenated into a target file located at `dist/assets/app/js/app-templates-<%= pkg.version %>.js`. During execution, `<%= pkg.version %>` is replaced by the value of the `version` attribute of the `package.json` file.

The JST compiler creates one function for each HTML template in the source directory. By default, each function is placed in an object named `JST` which is attached to the browser Window object. In the options section, the namespace attribute value is used to override the default `JST` object name. The `processName` attribute contains a function which is called once for each template found in the source directory. The function is passed a parameter containing the relative path of the source file. The function returns a string used as the attribute name of the compiled template function.

For example, we created a template named `navbar.html`. After the JST Grunt task is executed, the compiled template is available to the issue tracker application as the function `IssueTrackerTemplates.navbar(data)`.

Why is this important? Marionette applications use the `Marionette.Renderer` class's `render` function to process HTML templates into a DOM structure which is subsequently attached to the browser page. We must override the default implementation of the `render` function so that Marionette finds the precompiled template functions.

Rendering Precompiled Templates

Grunt compiles each template source file into a JavaScript function accessible via `IssueTrackerTemplates.templateName(data)`. The `data` parameter contains a JSON object which, if provided, is used by the function to generate dynamic HTML using the template.

By default, the Marionette `Renderer` logic expects HTML templates to be located in the `index.html` page and referenced by a unique `id` attribute. This works well for small applications, but when the application grows, placing all of the templates on the HTML page is impractical. The solution is to author templates in individual source files and precompile them into JavaScript. Since they are no longer found on the index page using an `id` attribute selector, we need to override the `Renderer.render` method.

At the top of the `Application.js` file, add the code below.

Override the Renderer

```
1 // Customize the Renderer to use Namespacing
2 Backbone.Marionette.Renderer.render = function(template, data) {
3   return IssueTrackerTemplates[template](data);
4 };
```

Now, the Marionette `Renderer` will use the namespace, `IssueTrackerTemplates`, and the template name to find the precompiled template function. The template function is called, passing the optional dynamic data. The HTML generated by the template function is returned by the `render` function.

Header Views

A view class is needed to manage the navbar template. To promote high cohesion, the JavaScript classes responsible for the application header are defined within a Marionette Module. To create

module definitions spanning multiple source files, we will employ the callback function definition, to define modules. Using callback function definitions, the classes that comprise a module may be declared in multiple source files. For instance, the module controller may be declared in one JavaScript file and the views in another. Using a structured approach to organize module classes improves the maintainability of the application.

A simple illustration of the module definition callback function approach is depicted below.

Module Definition Callback Function

```
1 IssueTrackerApp.module('Header', function(Header, IssueTrackerApp, Backbone, Mar\
2 ionette, $, _) {
3
4   // Header Module Logic
5
6 });
```

The callback function definition approach uses the `module` function on the Marionette application. The first parameter is the module name. The second parameter contains the module definition callback function. The callback function is invoked immediately. It receives a reference to the module being declared, a reference to the application, Backbone, Marionette, jQuery (\$), and Underscore (_). In the function body the module attributes and behaviors are defined. Module definitions may be split into separate source files, each invoking the module definition callback function. The first invocation of a module's callback function creates a new module instance within the Marionette application and subsequent callback functions augment the functionality of the module rather than creating a new module.

Let's create a module named *Header* using the callback function definition in two source files. To promote high cohesion in our application, create the module views in one source file and the module controller in a second source file. To keep the module source files organized in the source directory structure, the JavaScript for each module will be located in a distinct sub-directory.

In the `src/main/app/js/modules` directory, create a new sub-directory named `header`. In the `header` directory, create a new file named `HeaderViews.js`. Within the `HeaderViews.js` file add the content below.

HeaderView.js

```
1 IssueTrackerApp.module('Header',
2   function(Header, IssueTrackerApp, Backbone, Marionette, $, _) {
3
4     // Define the NavBarView Class
5     Header.NavBarView = Backbone.Marionette.ItemView.extend({
6
7       template: 'navbar'
8
9     });
10
11 });
```

The code begins with the module definition callback function. The `IssueTrackerApp` creates or, if it already exists, augments the header module.

Within the function, the navbar view class is declared. Note the syntax `Header.NavBarView` is used rather than `var Header.NavBarView`. By omitting the `var` keyword the `NavBarView` class is implicitly declared as a public member of the `Header` module allowing the `NavBarView` class to be accessed and instantiated anywhere the `Header` module is accessible.

The `NavBarView` class has a single attribute, `template`, whose value is the name of the HTML template to be managed by this view. The JST task in the Grunt build process removes the `.html` extension from template source files as it compiles templates into named JavaScript functions. Compiled template functions are referenced by the name of the source file minus the extension.

Header Controller

The module controller class is responsible for the orchestration of all activities within the module as well as module lifecycle events. In the `src/main/app/js/modules/header` directory, create a new file named `HeaderController.js`. Within the file, add the content below.

HeaderController.js

```
1 IssueTrackerApp.module('Header',
2   function(Header, IssueTrackerApp, Backbone, Marionette, $, _) {
3
4     // Define the Controller for the Header module
5     var HeaderController = Marionette.Controller.extend({
6
7       show: function() {
8         logger.debug("HeaderController.show");
9         var navBarView = new Header.NavBarView();
```

```
10     IssueTrackerApp.headerRegion.show(navBarView);
11   }
12
13   });
14
15   // Create an instance
16   var controller = new HeaderController();
17
18   // When the module is initialized...
19   Header.addInitializer(function() {
20     logger.debug("Header initializer");
21     controller.show();
22   });
23
24 });
```

The module definition callback function is employed once again to augment the overall Header module content.

The controller class, `HeaderController`, is declared using `var` to make it a private member of the Header module. Create a controller function named `show` to *show* the header to the application user. The function creates an instance of the `Header.NavBarView` class. Using the `IssueTrackerApp` parameter supplied via the module definition callback function, the controller accesses the `headerRegion`. The region's `show` function is called passing the navbar view object. The region manager assumes control, invoking the view's `show` method to render the template and appending the resulting HTML DOM to the parent element of the region. If the region had already shown a view, that view's `destroy` method would be called to clean up the old view before showing the new view.

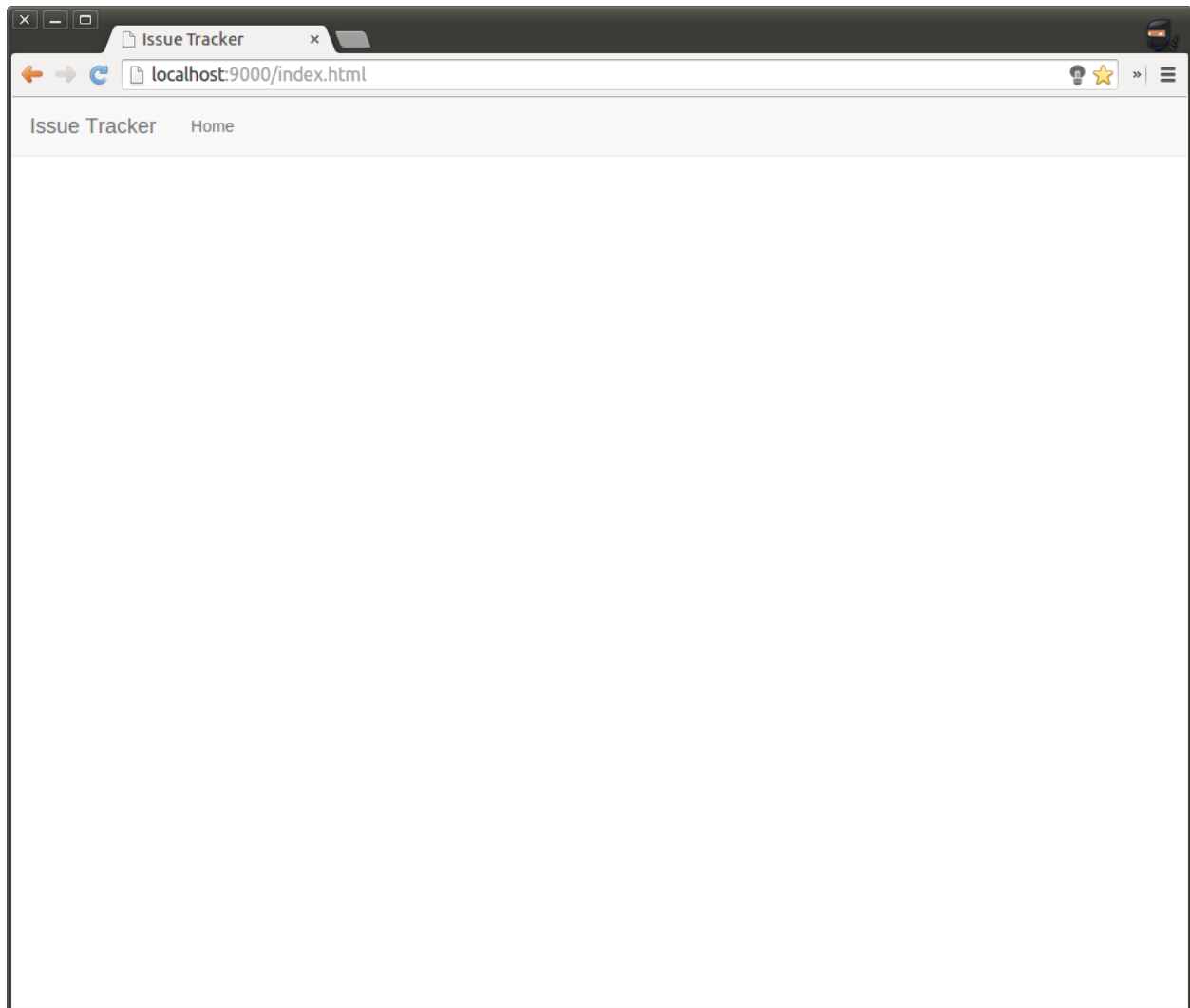
An instance of the `HeaderController` class is instantiated as private variable `controller`. The controller object may only be accessed by members of the Header module.

Finally, a module initializer is created. Module initializers are callback functions invoked either when the Marionette application is started, or, if the application has already been started, when the module is created using the `module` function on the application. Since the application uses the module definition callback function method to create modules, they are started when the application starts and all module initializers are immediately called. The navbar should be displayed to the user as soon as the application is started. The module initializer callback function invokes the `show` method on the controller to display the navbar in the header region of the page immediately upon application startup.

Running the Application

Let's test the Header module. Open a terminal window and navigate to the project base directory. Type `grunt run` and press *Enter* to start the local web server on port 9000. Open a browser and press

F12 to open the developer tools. In the browser's address bar type `http://localhost:9000/index.html` and press *Enter*. The browser page displays the navigation bar at the top of the page.



Header

In the console tab of the developer tools, application logging statements depict the flow of events.

Console Log

```
1 12:55:45 DEBUG - IssueTrackerApp.start
2 12:55:45 DEBUG - Header initializer
3 12:55:45 DEBUG - HeaderController.show
4 12:55:45 DEBUG - Backbone.history.start
```

Once again, the application starts when jQuery receives the document ready event. Now though, the Header module initializer is invoked, calling the `show` method on the HeaderController. Finally,

the application's start event listener callback method is invoked, starting the Backbone history. This sequence of log statements illustrates that all Marionette modules are started and their initializers are invoked prior to the application's start event being published. If needed, module functionality may be used in an application's start event listener callback function.

The Footer Module

The footer module displays static content in the footer region of the application. The footer module is nearly identical to the header module.

Footer Template

In the `src/main/app/templates` directory, create a new HTML template named `footer.html` and include the content below.

footer.html

```
1 <p>
2   A LeanStacks Solution
3 </p>
```

Update the CSS file to style the footer. Let's add padding, margin, and border to create separation between the footer and the main content. Add the footer rule set to the `app.css` file as shown below.

app.css

```
1  /* Project Styles */
2  /* Use bootstrap styles first and override only if necessary. */
3
4  /* body padding for http://getbootstrap.com/components/#navbar-fixed-top */
5  body {
6      padding-top: 70px;
7  }
8
9  footer {
10     padding-top: 40px;
11     padding-bottom: 40px;
12     margin-top: 100px;
13     color: #777;
14     text-align: center;
15     border-top: 1px solid #e5e5e5;
16 }
```

Footer Views

Next create the footer module and a view class responsible for managing the footer template. Begin by creating a sub-directory named `footer` in the `src/main/app/js/modules` directory. The footer directory should be a sibling of the header directory. In the footer directory, create a source file named `FooterViews.js` to contain the module view classes. Open the file and add the code shown below.

FooterViews.js

```
1 IssueTrackerApp.module('Footer', function(Footer, IssueTrackerApp, Backbone, Mar\
2 ionette, $, _) {
3
4   // Define the FooterView Class
5   Footer.FooterView = Backbone.Marionette.ItemView.extend({
6
7     className: 'container-fluid',
8     template: 'footer'
9
10  });
11
12  });
```

Create the Footer module using the module definition callback function. The FooterView class contains the `template` attribute similar to the NavBarView in the Header module; however, it also contains a new attribute, `className`. All Marionette views have a parent HTML tag stored in the `e1` attribute of the view class. By default, views use the `div` tag. When a view renders its template generating HTML markup, that markup is appended to the `e1` attribute, or `div` tag. Use the `tagName` attribute to change the type of HTML tag used by the view. Use the `className` attribute to specify one or more CSS classes for the view's parent HTML tag. Using the default `tagName` and `className: 'container-fluid'`, the FooterView's parent element is `<div class="container-fluid"></div>`. The HTML markup rendered from the footer template is appended to the parent element when the view's `show` function is called.

Footer Controller

Let's create the controller class for the Footer module. In the `src/main/app/js/modules/footer` directory, create a new file named `FooterController.js`. Open the file and add the code depicted below.

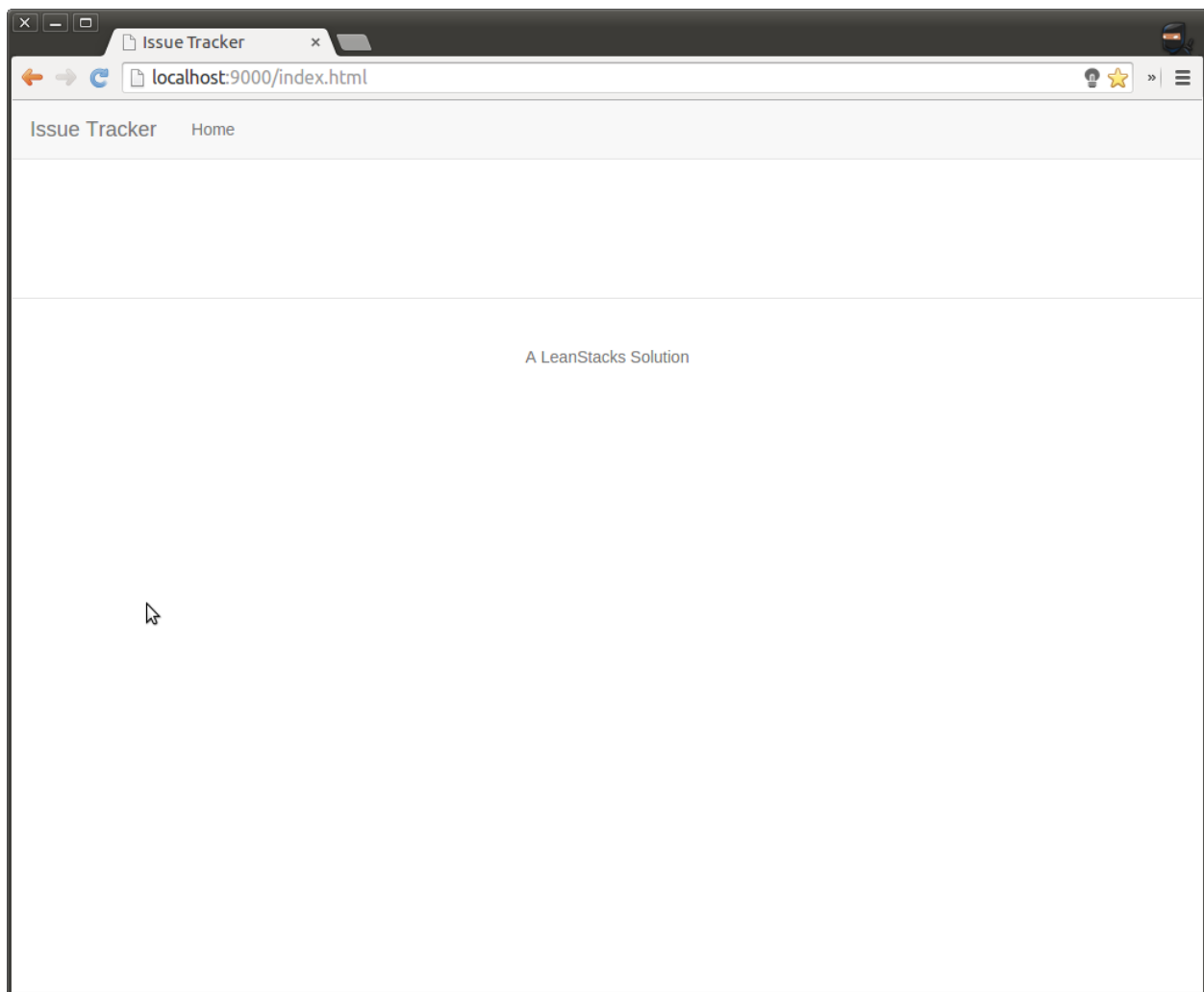
FooterController.js

```
1 IssueTrackerApp.module('Footer', function(Footer, IssueTrackerApp, Backbone, Mar\
2 ionette, $, _) {
3
4   // Define the Controller for the Footer module
5   var FooterController = Marionette.Controller.extend({
6
7     show: function() {
8       logger.debug("FooterController.show");
9       var footerView = new Footer.FooterView();
10      IssueTrackerApp.footerRegion.show(footerView);
11    }
12
13  });
14
15  // Create an instance
16  var controller = new FooterController();
17
18  // When the module is initialized...
19  Footer.addInitializer(function() {
20    logger.debug("Footer initializer");
21    controller.show();
22  });
23
24  });
```

Create a controller class for the Footer module which contains a show function. After creating an instance of the controller, add a module initializer callback function to call the show function on the controller object when the module is started by the application.

Running the Application

Let's test the Footer module. Open a terminal window and navigate to the project base directory. Type `grunt run` and press *Enter* to start the local web server on port 9000. Open a browser and press *F12* to open the developer tools. In the browser's address bar type `http://localhost:9000/index.html` and press *Enter*. The browser displays the navigation bar at the top of the page and the footer at the bottom separated by the spacing that we added to the application CSS.



Header and Footer

In the console tab of the developer tools, application logging statements depict the flow of events.

Console Log

```
1 12:55:45 DEBUG - IssueTrackerApp.start
2 12:55:45 DEBUG - Footer initializer
3 12:55:45 DEBUG - FooterController.show
4 12:55:45 DEBUG - Header initializer
5 12:55:45 DEBUG - HeaderController.show
6 12:55:45 DEBUG - Backbone.history.start
```

As before, the application is started when jQuery receives the document ready event. Now though, we see that the Header and Footer module initializers are invoked, calling the show methods on their respective controllers. Finally, the application's start event listener callback method is invoked, starting the Backbone history.

Organizing the Application

In this chapter, we have added many new files to the issue tracker user interface project. The `lib` directory contains several third party dependencies. The `src` directory now contains HTML templates and JavaScript source files. Before we enhance the application further, let's review the structure of the project thus far.

Project Structure

```
issue-tracker-ui
├─ Gruntfile.js
├─ lib
│   ├─ backbone-1.2.1.min.js
│   ├─ backbone.marionette-2.4.2.min.js
│   ├─ backbone.marionette.map
│   ├─ backbone-min.map
│   ├─ bootstrap-3.3.5
│   │   ├─ css
│   │   │   ├─ bootstrap.css
│   │   │   ├─ bootstrap.css.map
│   │   │   ├─ bootstrap.min.css
│   │   │   ├─ bootstrap-theme.css
│   │   │   ├─ bootstrap-theme.css.map
│   │   │   └─ bootstrap-theme.min.css
│   │   ├─ fonts
│   │   │   ├─ glyphicons-halflings-regular.eot
│   │   │   ├─ glyphicons-halflings-regular.svg
│   │   │   ├─ glyphicons-halflings-regular.ttf
│   │   │   ├─ glyphicons-halflings-regular.woff
│   │   │   └─ glyphicons-halflings-regular.woff2
│   │   └─ js
│   │       ├─ bootstrap.js
│   │       └─ bootstrap.min.js
│   └─ fontawesome-4.3.0
│       ├─ css
│       │   ├─ font-awesome.css
│       │   └─ font-awesome.min.css
│       └─ fonts
│           ├─ FontAwesome.otf
│           ├─ fontawesome-webfont.eot
│           ├─ fontawesome-webfont.svg
│           ├─ fontawesome-webfont.ttf
│           └─ fontawesome-webfont.woff
```



```
|   ├── jquery-2.1.4.min.js
|   ├── jquery-2.1.4.min.map
|   ├── json2.js
|   ├── log4javascript-1.4.13.js
|   ├── underscore-1.8.3.min.js
|   └── underscore-min.map
└── package.json
    └── src
        └── main
            └── app
                ├── css
                │   └── app.css
                ├── index.html
                ├── js
                │   ├── Application.js
                │   └── modules
                │       ├── footer
                │       │   ├── FooterController.js
                │       │   └── FooterViews.js
                │       └── header
                │           ├── HeaderController.js
                │           └── HeaderViews.js
                └── templates
                    ├── footer.html
                    └── navbar.html
```

A

The Road Ahead

In the next chapter, we will explore Marionette's messaging capabilities that allow the user interface application components to exchange information. Then we will use messaging as we enhance the application to integrate with the issue tracker web services.

Marionette Messaging

The Marionette Application class has a built-in messaging system to facilitate communication between the components of an application. Marionette uses the [Backbone.Wreqr](https://github.com/marionettejs/backbone.wreqr)⁴⁹ library to perform the communication. The core of the messaging system is the radio channel from Backbone.Wreqr, offering three messaging mechanisms.

Event Aggregator

The event aggregator is available through the `vent` property of the Marionette Application class. The event aggregator extends the `Backbone.Events` class and provides the principal event management logic in the application. The event aggregator implements the publish-subscribe pattern. Components publish events to the aggregator. If other components need to take action when those events occur, they subscribe to the aggregator with a callback method to be invoked when the event is published.

For example, a user clicks a link to sign out of the application. The view that manages the signout HTML template is notified the user has clicked the sign out link. The view publishes a *signout* event to the aggregator.

Publish to the Event Aggregator

```
IssueTrackerApp.vent.trigger('signout');
```

Continuing the example, perhaps there is a class that manages the cookies used by the application and we need to remove the *user* cookie when the user signs out. The code subscribes to the event aggregator for *signout* events.

Subscribe to the Event Aggregator

```
IssueTrackerApp.vent.on('signout', function() {  
    // remove cookies  
});
```

It is also possible to send data when publishing events. The code below illustrates passing an object representing the user with the sign out event.

⁴⁹<https://github.com/marionettejs/backbone.wreqr>

Passing Data via Event Aggregator

```
var user = { firstname: 'Joe', lastname: 'User' };
IssueTrackerApp.vent.trigger('signout', user);

IssueTrackerApp.vent.on('signout', function(user) {
  // remove cookies
  // do something with user
});
```

Commands

Commands are used to ask another component to execute some functionality asynchronously. Commands are a form of point-to-point remote communication similar to a unidirectional queue. Unlike event aggregator messages which may have many recipients, command messages have a single recipient, called a command handler. Like the event aggregator, there is no response returned from the function that handles the command.

Commands are available through the `commands` property of the `Marionette Application` class. The `execute` function is used to request that a command be run and data may be passed with the request. The `setHandler` function specifies the command handler callback function to be invoked when the command is executed.

For example, an application has logic that displays a user message. Commands can be used to reuse this code throughout the application, even though the component that needs to display a message does not have direct access to the user message display component.

Display Messages via Commands

```
1 // Set the handler for 'message:display' commands
2 IssueTrackerApp.commands.setHandler('message:display', function(msg) {
3   alert(msg);
4 });
5
6 // Execute a command to display a message
7 IssueTrackerApp.commands.execute('message:display', 'Invalid Input');
8
9 // Shortcut notation to execute a command
10 IssueTrackerApp.execute('message:display', 'Invalid Input');
```

Like event aggregator messages, it is possible to execute commands with zero to many parameters.

Request Response

The Request Response messaging facility is used to synchronously retrieve data from another application component. This allows application components to be decoupled, meaning that a component need not have a direct reference to another component in order to obtain data from it. Request Response is a form of point-to-point remote communication similar to a bidirectional queue.

Request Response is available through the `reqres` property of the `Marionette.Application` class. Use the `request` function to *request* data from a remote application component. Use the `setHandler` function to specify the request handler callback function invoked when the request is received. The object returned by the callback function is delivered to the requesting component.

For example, a view in the application needs to obtain information about the current user to display a friendly message. The view doesn't need to know where the user object is stored or how to fetch it; that is not the responsibility of a view. The view uses a Request Response message to request the user object and the application component responsible for managing the user data handles the request.

Obtain the User via Request Response

```
1  // Set the handler for 'user:entity' requests
2  IssueTrackerApp.reqres.setHandler('user:entity', function(id) {
3    var user = {};
4    if( id ) {
5      // Fetch user from a web service using the supplied 'id'
6    } else {
7      // Get user from a Cookie
8      user = $.cookie('user');
9    }
10   return user;
11 });
12
13 // Request the user entity
14 var currentUser = IssueTrackerApp.reqres.request('user:entity');
15
16 // Shortcut notation to make requests
17 var currentUser = IssueTrackerApp.request('user:entity');
18
19 // Request the user entity passing the 'id'
20 var specificUser = IssueTrackerApp.request('user:entity', 1);
```

The request function accepts zero to many data parameters which are passed to the request handler

callback function. If a handler function declares parameters, but they are not included in the request, the parameters evaluate to undefined.



The Road Ahead

In the next chapter, we will integrate the user interface with the web services to fetch and display a list of Issues.

Integrating with Web Services

Previously, we created the Header and Footer modules to display static HTML markup in their respective regions on the HTML page. In this chapter we will create user interface components that call the *Get All Issues* web service and display the response.

Creating the Issue Entity

In software development, an *entity* is an object used to model a type of data. When we constructed the web services, we defined an Issue class with attributes for title, description, type, priority, and status. This Java class is the Issue entity for the web services application. We need to create a JavaScript Issue class in the user interface that not only houses the attributes of an issue, but also facilitates data exchange with the issue web services.

Global Entities Module

In Marionette applications, entity models may be made available to all application components, i.e. global, or only to the module in which the entity is defined. Most entities are used in multiple modules and, therefore, are defined in a global application scope. Occasionally, an entity is either specialized for use in a single module or the entity is only applicable to a single module. Entities that pertain to a single module are defined in a module scope.

Application components obtain global entity instances using request response messaging. Module scoped entities are defined within the context of the module in which they are used and, therefore, are accessed directly through private references to their entity classes.

To promote high cohesion in the application, global entities are created in a dedicated module. Let's begin by creating a new module directory named `entities` in the `src/main/app/js/modules` directory. In the `entities` module directory, create a new source file named `Issues.js`.

Entities Module

```
1 IssueTrackerApp.module('Entities', function(Entities, IssueTrackerApp, Backbone, \
2   Marionette, $, _) {
3
4   });
```

The Entities module is declared using the definition callback function. Next, let's create the Issue entity class.

Issue Model

Entity classes extend the [Backbone.Model](http://backbonejs.org/#Model)⁵⁰ class. A model contains attributes that represent the state of the entity and functions to get and set attribute values. Models also have functions to validate the object and keep it in sync with the server. Models use jQuery's AJAX API to synchronize their state with web services. As state changes occur, models publish [Backbone Events](http://backbonejs.org/#Events)⁵¹ such as add, change, destroy, etc. Marionette application components listen for relevant model events and automatically take action when they are published.

Let's add the Issue model class to the module definition function.

Issue Model

```
1  // Define the Model for an Issue entity
2  Entities.Issue = Backbone.Model.extend({
3
4  });
```

The Issue class is defined as a public member of the Entities module, permitting instantiation anywhere in the application.

Issue Collection

An entity collection manages a set of model objects. Collections extend the [Backbone.Collection](http://backbonejs.org/#Collection)⁵² class. They have functions to order and filter their models. Collections also have functions to add, change, or remove models and, when invoked, trigger the corresponding web services to be called to keep server data synchronized. Similar to model objects, collections publish events to communicate when changes occur to their data.

Issue Collection

```
1  // Define a Collection of Issue entities
2  Entities.IssueCollection = Backbone.Collection.extend({
3
4      model: Entities.Issue,
5
6      url: 'http://localhost:8080/issues'
7
8  });
```

⁵⁰<http://backbonejs.org/#Model>

⁵¹<http://backbonejs.org/#Events>

⁵²<http://backbonejs.org/#Collection>

The `IssueCollection` class is declared as a public member of the `Entities` module, permitting instantiation anywhere in the application.

The `model` attribute defines the model class whose objects are managed by the collection.

The `url` attribute is the base URL of the web services family that manages the server-side state represented by the model and collection. For a detailed description of the default RESTful web service URL mappings that Backbone uses, you may read the [Backbone.sync⁵³](http://backbonejs.org/#Sync) documentation.

Backbone sync maps CRUD operations to RESTful web services in this manner.

Verb	Context Path	Action
GET	/collection	Retrieves a collection of entity models
GET	/collection/id	Retrieves a model for the specified ID
POST	/collection	Create a new model
PUT	/collection/id	Updates the model for the specified ID
DELETE	/collection/id	Deletes the model for the specified ID

We have created one of the five Backbone.sync RESTful web services. The *Get All Issues* web service returns a collection of entity models and is mapped to HTTP GET requests sent to the context path `/issues`. As this book progresses, we will create all of the CRUD RESTful web services for the `Issue` entity.

Backbone supports a sixth web service URL mapping which is not covered in this book. The *PATCH* web service mapping is similar to *PUT*; however, only the model attributes which contain updated values are sent to the server rather than the entire model object.

Verb	Context Path	Action
PATCH	/collection/id	Updates selected attributes of the model for the specified ID

Controller

Although the *Entities* module does not have view classes or a router like a traditional Marionette module, it needs a controller class to orchestrate the activities on the entities within the module. Entity controller behaviors fetch data from web services, implement caching logic, and save data into cookies or local storage. Below the `IssueCollection` in the module definition callback function, let's define a class named `IssueEntityController`.

⁵³<http://backbonejs.org/#Sync>

Issue Entity Controller

```
1  // Define the Controller for the Issue Entity
2  var IssueEntityController = Marionette.Controller.extend({
3
4      getIssues: function() {
5          logger.debug("IssueEntityController.getIssues");
6          var issues = new Entities.IssueCollection();
7          var defer = $.Deferred();
8          issues.fetch({
9              success: function(data) {
10                  defer.resolve(data);
11              }
12          });
13          return defer.promise();
14      }
15
16  });
17
18  // Create an instance
19  var issueController = new IssueEntityController();
```

Within the `IssueEntityController`, the `getIssues` function instantiates a new `IssueCollection` and requests the data to populate it from the web service. Remember that AJAX calls are asynchronous. The jQuery `Deferred` object and `Promise` API are used when fetching data from the server. The `Promise` API wraps functionality and executes progress, success, and failure callback functions as the functionality executes and when it completes.

The `getIssues` function instantiates a `Deferred` object, then calls the `fetch` function on the issues collection to initiate the AJAX call to the *Get All Issues* web service. The `fetch` function accepts a hash object containing success and error callback functions which are invoked based upon the result of the AJAX operation. If the AJAX call returns successfully, the `Deferred.resolve` function is called which invokes any success callback functions attached to the `Deferred` object.

The `Deferred` object's `Promise` is returned by the `getIssues` function. The `Promise` object exposes an API that allows components to attach callback functions to it such as `done`, `fail`, `progress`, etc. However, the `Promise` object does not allow code to affect the outcome of the functionality wrapped by the original `Deferred` object by calling `resolve`, `reject`, etc. By returning the `Promise` object from the `getIssues` function, the calling component code may attach callback functions that will be invoked when the web service returns.

Request Handler

The request / response messaging facility is used to obtain global entity objects from any component of a Marionette application. This allows the global entities to be decoupled from the application modules that use them, promoting the reuse of common entity data and behaviors throughout the application.

Below the `IssueEntityController` class in the `Issue.js` file, define a function that handles `issue:entities` requests.

Issue Collection Request Handler

```
1  // Handle request for a Collection of Issue Entities
2  IssueTrackerApp.reqres.setHandler('issue:entities', function() {
3    logger.debug("Handling 'issue:entities' request");
4    return issueController.getIssues();
5  });
```

The request handler function calls the `IssueEntityController.getAllIssues` function and returns the Promise object to the client in the `reqres` response message.

Creating the Issue Manager Module

Let's create a new module that encapsulates all issue related activities. We will call it the *Issue Manager* module.

Begin by creating a new module sub-directory named `issuemanager` in the `src/main/app/js/modules` directory. In the `issuemanager` directory, create two new files: `IssueManagerController.js` and `IssueManagerViews.js`. Within each of those files, add the code below to declare the Issue Manager module to the `IssueTrackerApp`.

Issue Manager Module Definition

```
1  IssueTrackerApp.module('IssueManager', function(IssueManager, IssueTrackerApp, B\
2  ackbone, Marionette, $, _) {
3
4  });
```

The familiar module definition callback function is used to create the Issue Manager module in the Marionette application. These source files will contain the definitions for the Issue Manager views, controller, and router. First, however, we need to create the HTML templates to display the data in the `IssueCollection`.

Introducing the CompositeView

The Marionette library implements many common design patterns for Backbone applications. Several of those patterns are implemented as specialized View classes. In the Header and Footer modules, the most common Marionette view class, the `ItemView`, is utilized. `ItemViews` are used to manage the rendered display of a single item, or entity model. The `CollectionView` class iterates over, you guessed it, a Collection of entities and displays each of them in a `childView` which is an `ItemView`. The `CollectionView` automatically listens for events on the collection such as `add` or `remove` and creates or destroys `ItemView` child instances.

The `CompositeView` class extends `CollectionView` and is a hybrid of an `ItemView` and `CollectionView`. A `CompositeView` renders an `ItemView` that wraps the collection's child views. Just like `CollectionView`, each child view renders a single item in the collection. A simple example of a `CompositeView` is an unordered list where each item in the collection is rendered as a list item. The `CompositeView`'s main template will contain the `` tag. The child view template will contain the `Some item data goes here.` tag with attributes of the item dynamically rendered between the opening and closing `li` tags.

Let's begin the Issue Manager module by constructing the HTML templates needed for the composite view.

Creating the Templates

The `CompositeView` uses three templates to display the collection of issue entities. The list template is the wrapper for entire view. The list item template renders the HTML markup for a single issue in the collection. The empty list template is rendered by the `CompositeView` in lieu of the list item template when the collection does not contain any issue entities.

List Template

In the `src/main/app/templates` directory, create a new file named `issuelist.html`.

Open the `issuelist.html` template and add the HTML depicted below.

`issuelist.html`

```
1 <div class="row">
2   <div class="col-md-12">
3     <div class="page-header">
4       <h1>Issues</h1>
5     </div>
6   </div>
7 </div>
8
9 <div class="row">
```

```

10 <div class="col-md-12">
11   <table class="table table-hover">
12     <thead>
13       <tr>
14         <th class="col-md-1">Priority</th>
15         <th class="col-md-1">Type</th>
16         <th class="col-md-1">Status</th>
17         <th class="col-md-9">Title</th>
18       </tr>
19     </thead>
20     <tbody>
21     </tbody>
22   </table>
23 </div>
24 </div>

```

This template is used by the `CompositeView` class as a wrapper for the collection it manages. When this template is rendered it creates a heading row above the list of issues. The CSS classes `row` and `col-md-12` are part of the [Bootstrap grid system](http://getbootstrap.com/css/#grid)⁵⁴ that divides the page into 12 columns. In this case, a full width row is created to contain the view heading.

A second, full width row is created containing an empty HTML table. CSS classes from the [Bootstrap table styles](http://getbootstrap.com/css/#tables)⁵⁵ add a highlight to a table row when the mouse pointer hovers above it.

Item Template

In the `src/main/app/templates` directory, create a new file named `issuelistitem.html`.

Open the `issuelistitem.html` file and insert the HTML shown below.

`issuelistitem.html`

```

1 <td class="col-md-1">
2 <% switch( priority ) {
3   case "LOW": { %>
4     <i class="fa fa-arrow-down fa-fw text-success" title="Low"></i>
5     <% break;
6   }
7   case "MEDIUM": { %>
8     <i class="fa fa-chevron-up fa-fw text-primary" title="Medium"></i>
9     <% break;
10  }

```

⁵⁴<http://getbootstrap.com/css/#grid>

⁵⁵<http://getbootstrap.com/css/#tables>

```

11     case "HIGH": { %>
12     <i class="fa fa-arrow-up fa-fw text-danger" title="High"></i>
13     <% break;
14     }
15     } %>
16 </td>
17
18 <td class="col-md-1">
19 <% switch( type ) {
20     case "TASK": { %>
21     <i class="fa fa-check fa-fw" title="Task"></i>
22     <% break;
23     }
24     case "STORY": { %>
25     <i class="fa fa-book fa-fw text-primary" title="Story"></i>
26     <% break;
27     }
28     case "ENHANCEMENT": { %>
29     <i class="fa fa-wrench fa-fw text-muted" title="Enhancement"></i>
30     <% break;
31     }
32     case "BUG": { %>
33     <i class="fa fa-bug fa-fw text-danger" title="Bug"></i>
34     <% break;
35     }
36     } %>
37 </td>
38
39 <td class="col-md-1">
40 <% switch( status ) {
41     case "OPEN": { %>
42     <span class="label label-primary">Open</span>
43     <% break;
44     }
45     case "IN_PROGRESS": { %>
46     <span class="label label-warning">In Progress</span>
47     <% break;
48     }
49     case "DONE": { %>
50     <span class="label label-success">Done</span>
51     <% break;
52     }

```

```
53     } %>
54 </td>
55
56 <td class="col-md-9">
57     <%= title %>
58 </td>
```

This template is used by the `CompositeView`'s child view class to render one table row for each issue in the collection.

Thus far, the HTML markup in the templates has been static. The issue list item template introduces dynamic markup using [Underscore template instructions](http://underscorejs.org/#template)⁵⁶. Underscore template instructions are bracketed by less than and greater than symbols and percent signs like `<% some instruction %>`.

The issue list item template contains four table column, `td`, tags. The first three table columns represent the issue priority, type, and status respectively. Underscore template logic is employed to inject JavaScript case statements into each table column. The conditional logic customizes the HTML markup depending upon the attribute value. Font Awesome icons and Bootstrap font colors are varied to create different visual cues for each attribute value.

The fourth table column contains Underscore syntax in the format `<%= attributeName %>`. Adding the equal sign immediately after the first percent sign tells Underscore to insert the value of the attribute whose name is contained in the template instruction. The technical name for this template instruction is *variable interpolation*. If the attribute value needs to be escaped, use `<%= -attributeName %>`. As the `CompositeView` iterates over the collection of issues, it creates an instance of the list item child view class for each issue model. The list item child view's template dynamically renders the Issue model attributes for priority, type, status, and title into table columns.

Note that the template contains the `td` tags for each column in the table, but the `tr` tag is missing. In a later section, we will see how the `tr` tag is specified within the list item view class that renders this template.

Empty Template

In the previous section, we created the template used to render each issue in the collection returned by the server. But what happens if the server does not return any issues? Displaying an empty table to the user may be confusing. We need to tell users that everything is OK, there are simply no issues to list.

In the `src/main/app/templates` directory, create a new template named `issuelistempty.html`.

Open the `issuelistempty.html` file and add the HTML depicted below.

⁵⁶<http://underscorejs.org/#template>

issuelistempty.html

```
1 <td colspan="4" class="col-md-12">No issues to display.</td>
```

The empty list template is rendered inside the CompositeView's wrapper template. Therefore, the empty list template contains a single table column tag spanning the four columns of the table. The template displays a static message informing the user that no issues were found.

Creating the Views

Each of the three templates requires a View class to render it. We will create an ItemView class to represent an the individual list item, that is, the view for a single issue entity model within the collection. Likewise, we need an ItemView class to represent the empty list view. Finally, we will create a CompositeView class to render the wrapper template and manage the lifecycle for the collection of ItemView instances representing each issue in the collection or the single ItemView representing an empty collection.

Empty List View

Open the IssueManagerViews.js source file. Inside the module definition callback function, enter the code below to create the IssueListEmptyView class as a member of the IssueManager module.

Issue List Empty View

```
1 // Define the View for an empty List of Issues
2 IssueManager.IssueListEmptyView = Backbone.Marionette.ItemView.extend({
3
4     tagName: 'tr',
5
6     template: 'issuelistempty'
7
8 });
```

The template attribute contains the name of the empty list template source file minus the .html extension.

After a Marionette view renders the HTML DOM from their HTML template, the view appends the markup to it's parent HTML element. The default parent HTML tag is div. To override the default parent tag, supply an alternate HTML tag name in the tagName attribute. The HTML templates for the empty list and the list item contain one or more table column, td, tags without a table row, tr, tag surrounding them. When the IssueListEmptyView renders the empty list template, the resulting markup is appended to a table row, tr, tag.

List Item View

Open the `IssueManagerViews.js` source file. Below the `IssueListEmptyView` class, create the class definition for a list item.

Issue List Item View

```
1  // Define the View for a single Issue in the List
2  IssueManager.IssueListItemView = Backbone.Marionette.ItemView.extend({
3
4      tagName: 'tr',
5
6      template: 'issuelistitem'
7
8  });
```

The `IssueListItemView` class extends Marionette's `ItemView`. The list `CompositeView` will instantiate one `IssueListItemView` for each issue entity model in the collection of issues supplied to the `CompositeView`. The `IssueListItemView` uses an issue entity model instance to render the dynamic HTML in the template. Similar to the empty list view, the list item view overrides the `tagName` attribute, setting the view's parent element to the `tr` tag.

List View

Below the class definitions for `IssueListEmptyView` and `IssueListItemView`, create the class for the list `CompositeView`.

Issue List Composite View

```
1  // Define the View for a List of Issues
2  IssueManager.IssueListView = Backbone.Marionette.CompositeView.extend({
3
4      emptyView: IssueManager.IssueListEmptyView,
5
6      childView: IssueManager.IssueListItemView,
7
8      childViewContainer: 'tbody',
9
10     className: 'container-fluid',
11
12     template: 'issuelist'
13
14 });
```

The `IssueListView` extends Marionette's `CompositeView` class. `CompositeViews` manage multiple `ItemView` classes in addition to their wrapper template, therefore, they require a few additional attributes to configure the view behavior.

The `emptyView` attribute contains the `ItemView` class used when the `CompositeView`'s collection does not contain any models.

The `childView` attribute configures the `ItemView` class used to render an individual model from the `CompositeView`'s collection. The `childViewContainer` attribute contains a DOM selector which tells the `CompositeView` where to append the rendered HTML from the child views or the empty view. The selector must match a HTML DOM element within the `CompositeView`'s template. The `childViewContainer` attribute value is set to `tbody`, instructing the `IssueListView` to append the child view or empty view HTML to the `tbody` tag of the `issuelist` template.

The `tagName` attribute is omitted and, therefore, the `issuelist` template will be appended the default `div` parent tag. The `className` attribute is used to add one or more CSS classes to the `class` attribute of the view's parent tag. Therefore, the parent tag to which the wrapper template is appended is `<div class="container-fluid"> </div>`.

Creating the Module Controller Components

The issue manager module has more complexity than the header and footer modules. In addition to a controller class, we will create a router to listen for changes to the URL hash value in the browser address bar. We also need to create a command handler allowing application components outside of the issue manager module to request that the issue list be shown.

Router

Open the `IssueManagerController.js` source file. Inside the module definition callback function, use the code depicted below to create the `IssueManagerRouter` class within the `IssueManager` module definition callback function.

Issue Manager Router

```
1  // Define the AppRouter for the IssueManager module
2  var IssueManagerRouter = Marionette.AppRouter.extend({
3
4      appRoutes: {
5          'issues': 'list'
6      }
7
8  });
```

The `IssueManagerRouter` class extends the `Marionette AppRouter` class. Notice that the router is defined using the `var` keyword making it private within the Issue Manager module. Routers should never be instantiated or called directly by external components.

Routers listen for changes to the URL hash value. The URL hash value is the value after the pound sign, #.



Basic URL Format

`http://host[:port]/[path/]page.html[#hash][?param=value]` This basic URL format illustrates the common pieces of a URL in a single page application. The optional pieces are surrounded with square brackets.

When instantiated, a router is supplied an instance of a module controller. The `appRoutes` attribute contains a hash object. The object keys are URL hash values. Each URL hash value is mapped to a controller function. When a URL hash value matches one managed by a router, it invokes the named function on the controller.

Controller

Open the `IssueManagerController.js` file. After the `IssueManagerRouter` class, create the `IssueManagerController` class shown in the code snippet below.

Issue Manager Controller

```

1  // Define the Controller for the IssueManager module
2  var IssueManagerController = Marionette.Controller.extend({
3
4      list: function(collection) {
5          logger.debug("IssueManagerController.list");
6
7          var displayListView = function(issueCollection) {
8              var listView = new IssueManager.IssueListView({
9                  collection: issueCollection
10             });
11
12             logger.debug("Show IssueListView in IssueTrackerApp.mainRegion");
13             IssueTrackerApp.mainRegion.show(listView);
14         };
15
16         if(collection) {
17             displayListView(collection);
18         } else {
19             var fetchingIssues = IssueTrackerApp.request('issue:entities');
```

```
20     $.when(fetchingIssues).done(function(issues) {
21         displayListView(issues);
22     });
23 }
24 }
25
26 });
27
28
29 // Create an instance
30 var controller = new IssueManagerController();
```

The controller's `list` function orchestrates the activities necessary to display a list of issues in the application's main region. The function receives a single, optional parameter: `collection`. If present, the `list` function uses the contents of the `collection` parameter rather than fetching a collection from the server.

The `displayListView` function encapsulates the logic to prepare and show view objects within the application. The function accepts a single parameter containing the collection of issue entities to be listed. The `displayListView` function constructs an `IssueListView` object providing it the `issueCollection`. The `listView` is shown in the application's main region.

Following the `displayListView` function declaration, we determine if the `list` function was passed a collection parameter. If the `collection` parameter exists, the `displayListView` function is called using the collection parameter. If the collection parameter is undefined, a request message is sent to retrieve a collection of issues. The request handler function that we created in the Entities module receives this request and returns the Deferred object's Promise stored in the variable `fetchingIssues`. The jQuery `when` function listens for changes to the Deferred object. When the server responds and the Deferred object's `resolve` method is called in the Entities module, the Promise's `done` callback function is invoked. The `displayListView` function is called using the collection of issues returned by the server.



Avoiding Unnecessary AJAX Requests

At this time, the `list` function's `collection` parameter will always be undefined. However, as we add the capability to view and edit issues from the list, the collection parameter will be passed to the `list` function preventing unnecessary web service requests.

Module_INITIALIZER

Create a module initializer callback function for the Issue Manager module. Open the `IssueManagerController.js` source file and insert this code snippet after the `IssueManagerController` class definition.

Issue Manager Initializer

```
1  // When the module is initialized...
2  IssueManager.addInitializer(function() {
3      logger.debug("IssueManager initializer");
4      var router = new IssueManagerRouter({
5          controller: controller
6      });
7  });
```

The module initializer function is invoked when the Marionette application is started. A router instance is created and provided with a reference to the module controller.

Command Handler

We need to provide a way for external application components to trigger the display of the issue list. It is possible for other components to change the URL hash to `issues`, thus causing the `IssueManagerRouter` to show the issue list; however, routers are not meant to be used to trigger application flow of control. They are designed to allow users to bookmark URLs to return to a specific application state, or view.

It is preferable to use messaging to request view transitions from external components. The commands messaging mechanism facilitates this style of point-to-point unidirectional communication.

Insert the code snippet shown below after the issue manager module initializer.

List Issues Command Handler

```
1  // Handle application commands...
2  IssueTrackerApp.commands.setHandler('issuemanager:list', function(collection) {
3      logger.debug("Handling 'issuemanager:list' command");
4      IssueTrackerApp.navigate('issues');
5      controller.list(collection);
6  });
```

The command handler function calls the `navigate` function on the `IssueTrackerApp` to update the browser's URL hash value to `#issues`. The browser will automatically record the URL change in its history. Then the `list` controller function is called to display the issue list on the page. The command handler optionally receives a collection of issue models to display in the list which it passes to the controller function.

Listing Issues at Startup

Unlike the Header and Footer modules, the issue list is not immediately shown using the module initializer. We don't want to presume that the list of issues should be shown when the application starts. Suppose a user accesses the application with a bookmarked route, the application should show the view associated with that route rather than always showing the list of issues.

Let's create two utility functions to get and set the value of the URL hash. Open the `Application.js` source file and insert the two functions between the application declaration and the region definitions.

Navigation Utility Functions

```
1 // Create the Application
2 window.IssueTrackerApp = new Backbone.Marionette.Application();
3
4 // Navigate to a route
5 IssueTrackerApp.navigate = function(route, options) {
6     logger.debug("IssueTrackerApp.navigate route:" + route);
7     options = options || {};
8     Backbone.history.navigate(route, options);
9 };
10
11 // Retrieve the current route
12 IssueTrackerApp.getCurrentRoute = function() {
13     return Backbone.history.fragment;
14 };
15
16 // Create the top-level Regions
17 IssueTrackerApp.addRegions({
18     headerRegion : '#header-region',
19     mainRegion   : '#main-region',
20     dialogRegion : '#dialog-region',
21     footerRegion : '#footer-region'
22 });
```

The `navigate` and `getCurrentRoute` functions are highlighted in the excerpt above. Both functions are added to the `IssueTrackerApp` object making them available throughout the Marionette application. The `navigate` function updates the URL hash route in the browser's address bar and stores the updated route in the browser history. The `getCurrentRoute` function retrieves the current route value from `Backbone.history`.

Thus far, the Marionette application start callback function initializes the Backbone History component. Update the function to determine if the application has been started with a specific route. If there is no route present, execute the command handler to display the issue list.

Application Start Callback

```
1 // Application start Callback Function
2 IssueTrackerApp.on('start', function(options) {
3   // Initialize the Router
4   logger.debug("Backbone.history.start");
5   Backbone.history.start();
6
7   // Launch the Issue List
8   if(IssueTrackerApp.getCurrentRoute() === '') {
9     IssueTrackerApp.execute('issuemanager:list');
10  }
11 });
```

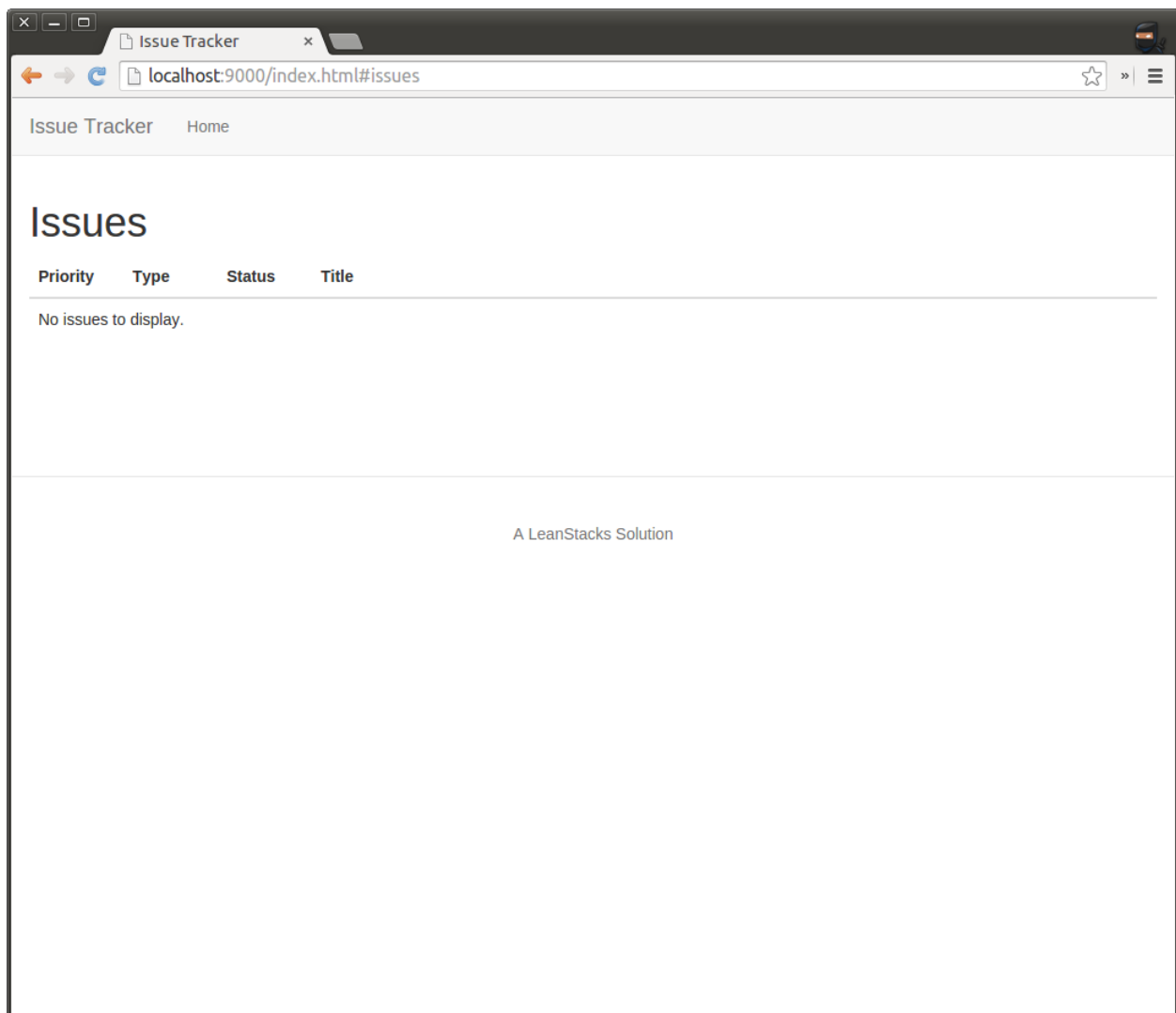
Running the Application

Let's test the Issue Manager module. The user interface now makes web service requests. We need to run both the web services and user interface applications.

Open a terminal window and navigate to the web services project base directory. Type `mvn spring-boot:run` and press *Enter* to start the web services on port 8080.

Open a second terminal window, or tab, and navigate to the user interface project base directory. Type `grunt run` and press *Enter* to start the local web server on port 9000.

Open a browser and press F12 to open the developer tools. In the browser's address bar type `http://localhost:9000/index.html` and press *Enter*. The browser page displays the navigation bar at the top of the page and the footer at the bottom. Between the header and footer, the issue manager module displays an empty issue list. Remember that we are using an in-memory HSQLDB database to persist issues. Since the database is empty, the *Get All Issues* web service returns an empty collection to the user interface. The issue list `ComponentView` displays the empty list view.



Issue Manager - List

In the console tab of the developer tools, the user interface application flow is depicted in logging statements.

Console Log

```
1 12:52:34 DEBUG - IssueTrackerApp.start
2 12:52:34 DEBUG - Footer initializer
3 12:52:34 DEBUG - FooterController.show
4 12:52:34 DEBUG - Header initializer
5 12:52:34 DEBUG - HeaderController.show
6 12:52:34 DEBUG - IssueManager initializer
7 12:52:34 DEBUG - Backbone.history.start
8 12:52:34 DEBUG - IssueManagerController.list
```

```
9 12:52:34 DEBUG - Handling 'issue:entities' request
10 12:52:34 DEBUG - IssueEntityController.getIssues
11 12:52:34 DEBUG - Show IssueListView in IssueTrackerApp.mainRegion
```

The application is started when jQuery receives the document ready event. The Header and Footer module initializers are invoked, calling the show methods on their respective controllers. Next, the application's start event listener callback method is invoked, starting the Backbone history and sending the `issuemanager:list` command which is received by the Issue Manager module. The issue collection is requested from the Entities module. After the data is returned by the server, the view is shown in the main region of the application.



The Road Ahead

We have learned the basic mechanics of creating web services to retrieve data from a database and return it as JSON. We have learned to create Marionette components to retrieve data from web services and display it in a single page application. In the next several chapters, we will enhance both our web services and user interface to create, update, and delete issues.

Part IV. CRUD

Creating Issues

We created the *Get All Issues* web service and the *Issue Manager* user interface module to list the issues returned by the web service. In this chapter, we will construct a web service to persist issues in the database and the user interface form to add issues.

Web Service

With the foundation of the web services project in place, creating a new web service requires little work. We need to add a method to the `IssueService` interface to create new `Issue` entities and code the implementation for that method in the `IssueServiceBean`. Finally, we will add a new web service request handler to the `IssueController`.

Service Interface

The service interface defines the contract for the service implementation class. Previously we authored the `findAll` method in the `IssueService` interface. Below it let's declare a new method to create `Issue` entities.

Issue Service - Create

```
1  /**
2   * Create a new Issue entity in the data repository.
3   * @param issue An issue entity to persist.
4   * @return The persisted issue entity.
5   */
6  Issue create(Issue issue);
```

The `create` method receives an `Issue` entity to be stored in the database and returns the persisted entity.

Service Implementation

The service implementation class contains the functionality for methods declared in the service interface. Add the implementation for the `create` method that we declared on the `IssueService` interface. Open the `IssueServiceBean` and insert the following logic below the `findAll` method.

Issue Service Implementation - Create

```
1      @Override
2      public Issue create(Issue issue) {
3          logger.info("> create");
4
5          // Set default attribute values
6          issue.setStatus(IssueStatus.OPEN);
7
8          if (issue.getPriority() == null) {
9              issue.setPriority(IssuePriority.MEDIUM);
10         }
11
12         // Persist the issue entity
13         Issue persistedIssue = issueRepository.save(issue);
14
15         logger.info("< create");
16         return persistedIssue;
17     }
```

The create method implementation sets the default value for the status attribute to the first value in the workflow of an Issue. If the IssueService client neglected to set the value for the priority attribute, the service sets the value to medium.

The Spring Data framework implements all of the logic to connect to the database, generate SQL insert statements, and store the contents of the Issue entity in the database. The save method returns the persisted version of the Issue entity. The value of the id attribute is populated with the primary key identifier generated by the database. The method returns the persisted Issue entity to the service client.

Controller

The controller class implements the RESTful web service request handler methods. Let's code the request handler method and annotations that will receive requests to create Issue entities. Open the IssueController class and add this code after the getAllIssues request handler method.

Issue Controller - Create

```
1  @RequestMapping(  
2      value = "/issues",  
3      method = RequestMethod.POST,  
4      consumes = MediaType.APPLICATION_JSON_VALUE,  
5      produces = MediaType.APPLICATION_JSON_VALUE)  
6  public ResponseEntity<Issue> createIssue(@RequestBody Issue issue) {  
7      logger.info("> createIssue");  
8  
9      Issue createdIssue = null;  
10     try {  
11         createdIssue = issueService.create(issue);  
12     } catch (Exception e) {  
13         logger.error("Unexpected Exception caught.", e);  
14         return new ResponseEntity<Issue>(HttpStatus.INTERNAL_SERVER_ERROR);  
15     }  
16  
17     logger.info("< createIssue");  
18     return new ResponseEntity<Issue>(createdIssue, HttpStatus.CREATED);  
19 }
```

The `createIssue` request handler method is annotated with `@RequestMapping`. Like the `getAllIssues` method, the context path is `/issues`; however, the POST HTTP method is used to create entities. The `RequestMapping` *consumes* element value is JSON, telling the Spring Framework to map this method to requests whose Content-Type HTTP header value is `application/json`. Similarly, the *produces* element instructs the Spring Framework that requests must have an `Accepts` HTTP header value of `application/json`.

The `@RequestBody` annotation is used to handle request data in a web service. Spring automatically uses the appropriate library to marshal the contents of the HTTP request body into the method parameter annotated with `@RequestBody`. The *consumes* value tells Spring to expect JSON in the HTTP request body. To marshal the JSON data into an `Issue` entity, Spring uses the [Jackson](http://wiki.fasterxml.com/JacksonHome)⁵⁷ library.

User Interface

In the last chapter, we created the major components of the issue manager module. Let's enhance the module to facilitate the creation of new issues. To do this, we need to create a HTML template with a form to collect user input and a view class for the template. We must enhance the controller to create the view and to save the issue to the server. Finally, the navigation bar must be enhanced with controls allowing a user to access the create issue form.

⁵⁷<http://wiki.fasterxml.com/JacksonHome>

Add Issue HTML Template

To create new issues, the HTML template must contain a form allowing users to enter information describing the issue. The Issue entity has five attributes: title, description, type, priority, and status. However, when creating a new issue, the status attribute is always OPEN. Therefore, we only need the user to enter values for the remaining four attributes.

The type and priority attributes contain values from an enumerated set. We could use a HTML select tag to collect this information, but we are going to use a Bootstrap button group instead. Button groups provide a distinct visual style for selecting one or more items from a small set of possible values.

In the `src/main/app/templates` directory, create a new file named `issueadd.html` and place the following HTML markup within it.

Add Issue HTML Template

```

1 <div class="row">
2   <div class="col-md-12">
3     <div class="page-header">
4       <h1>Add Issue</h1>
5     </div>
6   </div>
7 </div>
8
9 <div class="row"> <!-- begin outer row -->
10   <div class="col-md-12"> <!-- begin outer column -->
11
12     <form role="form">
13
14       <div class="row"> <!-- begin nested row -->
15
16         <div class="col-md-4"> <!-- begin 1st column -->
17
18           <div class="form-group">
19             <div class="btn-group btn-group-sm" data-toggle="buttons">
20               <label class="btn btn-default active">
21                 <input type="radio"
22                   name="type"
23                   value="TASK"
24                   autocomplete="off"
25                   checked>
26                 <i class="fa fa-check fa-fw"></i> Task
27               </input>
28             </label>

```

```

29     <label class="btn btn-default">
30         <input type="radio"
31             name="type"
32             value="STORY"
33             autocomplete="off">
34             <i class="fa fa-book fa-fw text-primary"></i> Story
35         </input>
36     </label>
37     <label class="btn btn-default">
38         <input type="radio"
39             name="type"
40             value="ENHANCEMENT"
41             autocomplete="off">
42             <i class="fa fa-wrench fa-fw text-muted"></i> Enhancement
43         </input>
44     </label>
45     <label class="btn btn-default">
46         <input type="radio"
47             name="type"
48             value="BUG"
49             autocomplete="off">
50             <i class="fa fa-bug fa-fw text-danger"></i> Bug
51         </input>
52     </label>
53 </div>
54 </div>
55
56 </div> <!-- end 1st column -->
57
58 <div class="col-md-3"> <!-- begin 2nd column -->
59
60     <div class="form-group">
61         <div class="btn-group btn-group-sm" data-toggle="buttons">
62             <label class="btn btn-default">
63                 <input type="radio"
64                     name="priority"
65                     value="LOW"
66                     autocomplete="off">
67                 <i class="fa fa-arrow-down fa-fw text-success"></i> Low
68             </input>
69         </label>
70         <label class="btn btn-default active">

```

```

71         <input type="radio"
72             name="priority"
73             value="MEDIUM"
74             autocomplete="off"
75             checked>
76         <i class="fa fa-chevron-up fa-fw text-primary"></i> Medium
77     </input>
78 </label>
79 <label class="btn btn-default">
80     <input type="radio"
81         name="priority"
82         value="HIGH"
83         autocomplete="off">
84     <i class="fa fa-arrow-up fa-fw text-danger"></i> High
85 </input>
86 </label>
87 </div>
88 </div>
89
90 </div> <!-- end 2nd column -->
91
92 </div> <!-- end nested row -->
93
94 <div class="form-group">
95     <label for="issue-title-input"
96         class="control-label">
97         Title
98     </label>
99     <input id="issue-title-input"
100         type="text"
101         name="title"
102         class="form-control"
103         placeholder="Enter a brief title..."
104         autofocus />
105 </div>
106
107 <div class="form-group">
108     <label for="issue-description-textarea"
109         class="control-label">
110         Description
111     </label>
112     <textarea id="issue-description-textarea"

```

```

113         name="description"
114         class="form-control"
115         rows="6"
116         placeholder="Describe the issue..." />
117     </div>
118
119     <button type="button"
120         class="btn btn-primary js-create"
121         data-loading-text="Creating...">
122         Create
123     </button>
124     <button type="button"
125         class="btn btn-link js-cancel">
126         Cancel
127     </button>
128
129 </form>
130
131 </div> <!-- end outer column -->
132 </div> <!-- end outer row -->

```

The template contains two, full-width rows. The first displays the *Add Issue* heading above the form. The second row contains the form.

Immediately inside the `form` tag, a new Bootstrap row begins. Bootstrap permits nesting rows within the columns of an existing row to further subdivide the contents of a column. Nested rows also have 12-columns, but the total width of the nested row is equal to that of the containing column. We want to have the button groups for the type and priority attributes in the same row rather than stacked vertically, so a nested row with two columns is created. Each column contains the markup for one button group.

Below the nested row are the form input tags for the issue title and description. At the bottom of the form there are two buttons. The button labeled *Create* triggers the form submission and the button labeled *Cancel* allows the user to return to the list of issues.

Note that the buttons have CSS classes prefixed with `js-`. These classes are not part of Bootstrap nor are they declared in the application CSS. Throughout the application CSS classes prefixed with `js-` are employed as selectors for JavaScript behavior listening to events on that HTML DOM element. When the view class is defined that manages this template, we will illustrate how the `js-*` classes are used.

Backbone.Syphon Library

When an HTML form is submitted, it can be cumbersome to extract the data from each form input tag and store it into an entity. The [Backbone.Syphon](#)⁵⁸ library serializes form data into a JSON object. The name attribute on each of the form inputs precisely matches the corresponding attribute in the Issue entity so that the JSON rendered by Backbone.Syphon may be used to create a new Issue model object when the form is submitted.

Download the Backbone.Syphon library and place the minified JavaScript file in the `lib` directory. Then update the `index.html` file to include the necessary script tag.

Add Backbone.Syphon to index.html

```
1  <script src="assets/lib/log4javascript-1.4.13.js"></script>
2  <script src="assets/lib/jquery-2.1.4.min.js"></script>
3  <script src="assets/lib/json2.js"></script>
4  <script src="assets/lib/underscore-1.8.3.min.js"></script>
5  <script src="assets/lib/backbone-1.2.1.min.js"></script>
6  <script src="assets/lib/backbone.syphon-0.5.0.min.js"></script>
7  <script src="assets/lib/backbone.marionette-2.4.2.min.js"></script>
8  <script src="assets/lib/bootstrap-3.3.5/js/bootstrap.min.js"></script>
9  <script src="assets/app/js/app-templates-0.1.0.js"></script>
10 <script src="assets/app/js/app-0.1.0.min.js"></script>
```

Issue Entity

In the last chapter, we created the Issue class which extends Backbone.Model. Thus far, the Issue class has only been used in conjunction with the IssueCollection class to fetch issues. We need to update Issue so that we can use the model directly to create new issue entities.

Open the `Issue.js` source file and update the Issue class as illustrated below.

Issue Entity

```
1  // Define the Model for an Issue entity
2  Entities.Issue = Backbone.Model.extend({
3
4      urlRoot: 'http://localhost:8080/issues',
5
6      defaults: {
7          status: 'OPEN'
8      },
9  });
```

⁵⁸<https://github.com/marionettejs/backbone.syphon>

```
10     validate: function(attrs, options) {
11         var errors = {};
12         if (!attrs.title) {
13             errors.title = 'Title is required.';
14         }
15         if (!attrs.description) {
16             errors.description = 'Description is required.';
17         }
18         if (!attrs.type) {
19             errors.type = 'Type is required.';
20         }
21         if (!attrs.priority) {
22             errors.priority = 'Priority is required.';
23         }
24         if (!_.isEmpty(errors)) {
25             return errors;
26         }
27     }
28
29     });
```

urlRoot

Just as the `url` attribute of a `Backbone.Collection` class defines the base URL for the web service family for the entity, the `urlRoot` property on a `Backbone.Model` serves the same purpose when a model is used outside a collection.



url and urlRoot

The value of a model's `urlRoot` attribute is usually the same as the `url` attribute of the collection class for that model. However, it is possible to set the `urlRoot` to a different value if the web service managing individual models has a different URL than the web service managing collections of models.

Default Attribute Values

The `defaults` attribute contains a hash object. The attributes of the hash object define the default values for model attributes. When a model instance is created, the default values are used if no value is supplied for those attributes. If a value is provided to the constructor for an attribute with a default value, the supplied value overrides the default.

Validation

The `validate` attribute contains an optional function used to validate the contents of a model object prior to communicating with the server. When the `save` function is called on a model, the Backbone framework first calls the `validate` function. If the `validate` function returns null, Backbone considers validation to be successful and sends the model to the server. If validation errors are present, the `save` function does not communicate with the server and returns `false` to indicate the model validation failed.

The `validate` function creates an object containing validation error messages. There is no prescribed format for the errors object. If validation errors are present, the function returns the errors object. Validation errors may be retrieved through the model object's `validationErrors` property.

Add Issue View

We need to create a view class to manage the `issueadd.html` template. Since the view does not manage a collection of objects and has no special layout requirements, it should extend the `ItemView` class.

Open the `IssueManagerViews.js` source file located in the `issuemanager` module directory. At the bottom of the module definition callback method, create the `IssueAddView` as shown below.

IssueAddView

```
1  // Define the View for Adding an Issue
2  IssueManager.IssueAddView = Backbone.Marionette.ItemView.extend({
3
4      className: 'container-fluid',
5
6      template: 'issueadd',
7
8      ui: {
9          createButton: 'button.js-create',
10         cancelButton: 'button.js-cancel'
11     },
12
13     events: {
14         'click @ui.createButton': 'onCreateClicked'
15     },
16
17     triggers: {
18         'click @ui.cancelButton': 'form:cancel'
19     },
20 }
```

```
21     onCreateClicked: function(e) {
22         logger.debug("IssueAddView.onCreateClicked");
23         e.preventDefault();
24         this.showProcessingState();
25         var data = Backbone.Syphon.serialize(this);
26         this.trigger('form:submit', data);
27     },
28
29     showProcessingState: function() {
30         var spinnerContent = '<i class="fa fa-circle-o-notch fa-spin"></i> ';
31         this.ui.createButton.button('loading');
32         this.ui.createButton.prepend(spinnerContent);
33     },
34
35     hideProcessingState: function() {
36         this.ui.createButton.button('reset');
37     },
38
39     onFormValidationFailed: function(errors) {
40         this.hideProcessingState();
41         this.hideFormErrors();
42         _.each(errors, this.showFormError, this);
43     },
44
45     showFormError: function(errorMessage, fieldKey) {
46         var $formControl = this.$el.find('[name="'+fieldKey+'"]');
47         var $controlGroup = $formControl.parents('.form-group');
48         var errorContent = '<span class="help-block js-form-error">'+errorMessage+\'
49 </span>';
50         $formControl.after(errorContent);
51         $controlGroup.addClass('has-error');
52     },
53
54     hideFormErrors: function() {
55         this.$el.find('.js-form-error').each(function() {
56             $(this).remove();
57         });
58         this.$el.find('.form-group.has-error').each(function() {
59             $(this).removeClass('has-error');
60         });
61     }
62 }
```

63 });

This class has a bit more complexity than the views we previously created. Since this view manages form submission and the display of validation errors, some additional logic is required.

The `ui` attribute contains a hash that maps elements from the rendered HTML template by their CSS selector. The `ui` hash provides a convenient mechanism for accessing frequently used HTML DOM elements throughout the view class. The `ui` elements may be referenced in the events hash using `@ui.elementName`. They may also be used throughout the view class in JavaScript using `this.ui.elementName`. In the `ui` hash we define UI elements for the two form buttons: `create` and `cancel`.

The events hash is derived from the `Backbone.View` class. Marionette's view classes have `Backbone.View` in their prototype chain. The events hash maps DOM events to a view function that is executed when the event occurs. The syntax of the events hash is `'event uiElement|CSSselector' : 'functionName'`. For example, `'click @ui.createButton' : 'onCreateClicked'` listens for *click* events on the *createButton* element defined in the `ui` hash, invoking the *onCreateClicked* function when the event is received. In the events hash we define a mapping for the *Create* button. When the create button is clicked, the *onCreateClicked* function is invoked.

The triggers hash is similar to the 'events' hash. However, when the mapped DOM event occurs, the view publishes a named event rather than invoking a function. When the *Cancel* button is clicked, the view triggers an event named `form:cancel`.

The remainder of the class contains functions that handle user events or modify the state of the view.

The *onCreateClicked* function handles user click events on the create button. The function calls the *showProcessingState* function which updates the HTML DOM to provide the user with visual feedback that the application is working. Next, the `Backbone.Syphon` library extracts the form data into a JSON object. Finally, the function triggers the `form:submit` event passing the serialized form data as a parameter. In the next section, we will see that the controller listens for this event and takes action to process the form data.

The *onFormValidationFailed* function is triggered by the controller if the Issue model validation function fails. The *onFormValidationFailed* function hides the processing state from the HTML DOM. Next, it calls *hideFormErrors* to remove any previously displayed validation errors. Finally, the function uses the `underscore` *each* function to iteratively call the *showFormError* function for each validation error in the `errors` object.

Controller

We need to add a function to the `IssueManagerController` to create an instance of `IssueAddView` and show it in the main application region. The controller also listens for `form:submit` events triggered by the view, creating new Issue entity models using the form data.

Open the `IssueController.js` source file. Below the `list` function, create a new function named `add` as illustrated below.

IssueManagerController - add Function

```

1      add: function() {
2          logger.debug("IssueManagerController.add");
3          var addIssueView = new IssueManager.IssueAddView();
4
5          // Handle 'form:cancel' event
6          addIssueView.on('form:cancel', function() {
7              logger.debug("Handling 'form:cancel' event");
8              IssueTrackerApp.execute('issuemanager:list');
9          });
10
11         // Handle 'form:submit' trigger
12         addIssueView.on('form:submit', function(data) {
13             logger.debug("Handling 'form:submit' event");
14             logger.debug("form data:" + JSON.stringify(data));
15             var issueModel = new IssueTrackerApp.Entities.Issue();
16             if(issueModel.save(data,
17                 {
18                     success: function() {
19                         IssueTrackerApp.execute('issuemanager:list');
20                     },
21                     error: function() {
22                         alert('An unexpected problem has occurred.');
```

An instance of `IssueAddView` is created. Using the `on` function, listen to the view object for `form:cancel` events. When the view triggers the event, the controller transitions the user to the issue list view.

A second event handler listens for `form:submit` events triggered by the view instance. The `form:submit` event handler receives a single parameter, `data`, containing the serialized form data. We create an instance of the `Issue` entity. Note that the class is referenced through the `Marionette` application itself because `Entities` module classes are not available directly from the `IssueManager` module.

Next, the `save` function is called on the model, passing the form data. The `save` function behavior is implemented by the `Backbone.View` class. First, the data is validated using the `Issue` model's `validate` function. If validation fails, the `save` function returns `false`. If validation succeeds, `Backbone` uses `jQuery` to send an `AJAX POST` request to the web service located at the model's `urlRoot` attribute and the function returns the `jQuery jqXHR` `AJAX` request object. The complete model, including any default attribute values, is serialized into the `AJAX` request body. The second parameter to the `save` function contains success and error callback functions passed to the `jQuery.ajax` function. The callback functions are invoked depending upon the outcome of the `AJAX` call. If the call is successful, the success function is invoked, displaying the issue list view. For now, if the call fails, we display a message in an alert box.

The `save` function is wrapped in an `if else` statement. Remember that `save` returns `false` if validation fails and returns the `jQuery jqXHR` `AJAX` request object when validation succeeds. Therefore, the `if` block is invoked when validation succeeds and the `else` block handles validation failures.

When validation fails, we use the `Marionette triggerMethod` function to invoke a function on the view object. The first parameter supplied to `triggerMethod` contains an event name like `form:validation:failed` followed by any parameters to pass to the function. `Marionette` converts the event name into a function name using camel case and prefixing it with `on`. Therefore, `addIssueView.triggerMethod('form:validation:failed', issueModel.validationError)` calls the function named `onFormValidationFailed` on `addIssueView` passing the validation errors object from `issueModel`.

After configuring the event handler logic, the controller function shows the `addIssueView` in the main region of the application.

Command Handler

To provide a means for application components to trigger the display of the add issue view, we need to add a command handler. In the `IssueController.js` file below the command handler for listing issues, add the code below.

Add Issue Command Handler

```
1 IssueTrackerApp.commands.setHandler('issuemanager:add', function() {  
2     logger.debug("Handling 'issuemanager:add' command");  
3     controller.add();  
4 });
```

The command handler calls the add method on the controller. The command handler does not contain the IssueTrackerApp.navigate() function because we do not want to change the URL hash in the browser address bar or save the add issue view in the browser history.

Navbar

To complete the add issue functionality, we need to create a way for users to access the new components. Let's create a button in the top navigation bar that shows the add issue form when clicked.

HTML Template

Open the navbar.html HTML template source file. Let's create the button that shows the add issue form and, also, a navigation link to the issue list.

navbar.html

```
1 <nav class="navbar navbar-default navbar-fixed-top" role="navigation">  
2     <div class="container-fluid">  
3         <!-- Brand and toggle get grouped for better mobile display -->  
4         <div class="navbar-header">  
5             <button type="button"  
6                 class="navbar-toggle collapsed"  
7                 data-toggle="collapse"  
8                 data-target="#bs-example-navbar-collapse-1">  
9                 <span class="sr-only">Toggle navigation</span>  
10                <span class="icon-bar"></span>  
11                <span class="icon-bar"></span>  
12                <span class="icon-bar"></span>  
13            </button>  
14            <a class="navbar-brand" href="#">Issue Tracker</a>  
15        </div>  
16  
17        <!-- Collect the nav links, forms, and other content for toggling -->  
18        <div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">
```



```

19     <ul class="nav navbar-nav">
20       <li class="dropdown">
21         <a href="#" class="dropdown-toggle" data-toggle="dropdown">
22           Issues <span class="caret"></span>
23         </a>
24         <ul class="dropdown-menu" role="menu">
25           <li>
26             <a href="#" class="js-nav" data-nav-target="issuemanager:list">
27               <i class="fa fa-list fa-fw"></i> List
28             </a>
29           </li>
30         </ul>
31       </li>
32     </ul>
33     <ul class="nav navbar-nav navbar-right">
34       <li>
35         <button type="button"
36           class="btn btn-default navbar-btn js-nav"
37           data-nav-target="issuemanager:add">
38           <i class="fa fa-plus fa-fw text-primary"></i> Issue
39         </button>
40       </li>
41     </ul>
42   </div><!-- /.navbar-collapse -->
43 </div><!-- /.container-fluid -->
44 </nav>

```

The complete `navbar.html` template source is shown above with the new tags highlighted. Bootstrap navbars contain one or more unordered lists of navigation links. Our navbar has two lists. The first list of links is located on the left side of the navigation bar. The second list uses the `navbar-right` CSS class which pulls the links to the right side.

Add a drop-down menu titled *Issues* with a single link. The link contains a CSS class, `js-nav`, which we will use in the UI element hash of the `NavBarView`. Create a custom data attribute named `data-nav-target` whose value is the Marionette command to be executed when the link is clicked.

The second unordered list is pulled to the right side of the navigation bar. To create a more distinctive navigation control, let's use a button instead of a standard link to access the add issue view.

View

In the `header` module, open the `HeaderViews.js` source file. Currently, the `NavBarView` displays a static HTML template. Let's modify the view class to send Marionette command messages when `js-nav` navigation elements are clicked.

NavBarView

```
1   Header.NavBarView = Backbone.Marionette.ItemView.extend({  
2  
3       template: 'navbar',  
4  
5       ui: {  
6           'navigation': '.js-nav'  
7       },  
8  
9       events: {  
10          'click @ui.navigation': 'onNavigationClicked'  
11      },  
12  
13      onNavigationClicked: function(e) {  
14          e.preventDefault();  
15          var commandName = $(e.target).attr('data-nav-target');  
16          IssueTrackerApp.execute(commandName);  
17      }  
18  });  
19  });
```

The complete NavBarView class is shown above with the new logic highlighted. The `ui` hash maps all HTML tags with the `js-nav` class to a UI element named `navigation`. The `events` hash ensures that each time a navigation UI element is clicked, the `onNavigationClicked` function is called.

The `onNavigationClicked` function extracts the value of the `data-nav-target` attribute value from the clicked UI element. The command is executed using the `IssueTrackerApp` resulting in the display of the desired view.

Running the Application

Let's test the new functionality. Open a terminal window and navigate to the web services project base directory. Type `mvn spring-boot:run` and press *Enter* to start the web services on port 8080. Open a second terminal window, or tab, and navigate to the user interface project base directory. Type `grunt run` and press *Enter* to start the local web server on port 9000.

Open a browser and press F12 to open the developer tools. In the browser's address bar type `http://localhost:9000/index.html` and press *Enter*. The browser page displays the updated navigation bar at the top of the page. The issue manager module displays an empty issue list. Click the add issue button in the top navigation bar. Complete the form and click the *Create* button. If everything works correctly, an AJAX POST request is sent to the server and the server returns a 201 status code

indicating the issue has been successfully created. The user interface returns to the issue list which now contains the new issue.

The screenshot shows a web browser window titled 'Issue Tracker' with the URL 'localhost:9000/index.html#issues'. The page has a header with 'Issue Tracker' and a dropdown menu for 'Issues', and a '+ Issue' button in the top right. The main content area is titled 'Add Issue' and contains a form with the following elements:

- Issue type buttons: Task (checked), Story, Enhancement, Bug.
- Priority buttons: Low (selected), Medium, High.
- Title input field: 'One Small Problem'.
- Description text area: 'I found a small problem in the application. To summarize, it's a small problem.'
- Buttons: 'Create' (blue) and 'Cancel' (light blue).

At the bottom of the page, it says 'A LeanStacks Solution'.

Issue Manager - Add

In the console tab of the developer tools, log statements depict the application flow.

Console Log

```
1 06:18:52 DEBUG - IssueTrackerApp.start
2 06:18:52 DEBUG - Footer initializer
3 06:18:52 DEBUG - FooterController.show
4 06:18:52 DEBUG - Header initializer
5 06:18:52 DEBUG - HeaderController.show
6 06:18:52 DEBUG - IssueManager initializer
7 06:18:52 DEBUG - Backbone.history.start
8 06:18:52 DEBUG - Handling 'issuemanager:list' command
```

```
9 06:18:52 DEBUG - IssueTrackerApp.navigate route:issues
10 06:18:52 DEBUG - IssueManagerController.list
11 06:18:52 DEBUG - Handling 'issue:entities' request
12 06:18:52 DEBUG - IssueEntityController.getIssues
13 06:18:52 DEBUG - Show IssueListView in IssueTrackerApp.mainRegion
14 06:19:01 DEBUG - Handling 'issuemanager:add' command
15 06:19:01 DEBUG - IssueManagerController.add
16 06:19:01 DEBUG - Show IssueAddView in IssueTrackerApp.mainRegion
17 06:19:24 DEBUG - IssueAddView.onCreateClicked
18 06:19:24 DEBUG - Handling 'form:submit' trigger
19 06:19:24 DEBUG - form data:{"type":"TASK","priority":"MEDIUM","title":"My Task",\
20 "description":"Complete this task."}
21 06:19:24 DEBUG - IssueManagerController.list
22 06:19:24 DEBUG - Handling 'issue:entities' request
23 06:19:24 DEBUG - IssueEntityController.getIssues
24 06:19:24 DEBUG - Show IssueListView in IssueTrackerApp.mainRegion
```

Lines 1 through 13 are the same as the last chapter. The Marionette application starts and displays the issue list. Line 14 occurs when the *Add Issue* button is clicked in the navbar. The application displays the add issue form in the main region of the application. Line 17 indicates that the *Create* button has been clicked, submitting the add issue form. The view triggers the `form:submit` event and the controller logs the data converted to JSON by Backbone.Syphon. After the AJAX call to the web service returns successfully, the controller executes the `list` function, displaying the issue list.



The Road Ahead

In the next chapter, we will enhance both the web services and user interface to view the details of an issue.

Viewing Issues

The applications have the capability to create and list issues. In this chapter, we will construct a new web service to find an issue entity in the database using the primary key identifier. We will enhance the user interface, adding the capability to retrieve the details of a single issue from the server and display the issue details to the user.

Web Services

Constructing a web service to find a single issue entity by primary key identifier requires changes to the business service interface, the service implementation bean, and the web service controller. The logic is similar to that used to find a collection of issue entities; however, the web service receives a parameter containing the primary key identifier of the desired issue.

Service Interface

The service interface defines the contract for the service implementation class. Previously the `findAll` method was declared in the `IssueService` interface. Just below it, let's define a new method to find a single `Issue` entity.

Issue Service - Find by Primary Key

```
1  /**
2   * Search the issue data repository for a single Issue entity by the primary
3   * key identifier.
4   * @param id An issue primary key identifier.
5   * @return An Issue entity or null if not found.
6   */
7  Issue find(Long id);
```

The `find` method receives a single parameter containing the primary key identifier of an `Issue` entity to be retrieved from the database. The method returns the `Issue` entity matching the supplied primary key value. If no entity exists in the database matching the primary key value, the method returns `null`.

Service Implementation

The service implementation class provides the functionality for methods declared on the service interface. We need to add the implementation for the `find` method that we declared on the `IssueService` interface. Open the `IssueServiceBean` and add this code snippet after the `findAll` method.

Issue Service Implementation - Find by Primary Key

```
1  @Override
2  public Issue find(Long id) {
3      logger.info("> find id:{", id);
4
5      Issue issue = issueRepository.findOne(id);
6
7      logger.info("< find id:{", id);
8      return issue;
9  }
```

The Spring Data framework implements all of the logic to connect to the database, generate SQL select statements, and retrieve the contents of the Issue entity from the database. The `findOne` repository method searches the database for an Issue entity whose primary key matches the supplied value. If found, the repository returns the Issue entity. If the database does not contain an Issue for the supplied primary key value, the repository returns `null`. The service implementation method returns precisely what was returned by the repository.

Controller

The controller class implements the RESTful web service request handler methods. Let's code the request handler method and annotations that will receive requests to find individual Issue entities by their primary key identifier. Open the `IssueController` class and add the request mapping method illustrated in the code snippet below.

Issue Controller - Get Issue

```
1  @RequestMapping(
2      value = "/issues/{id}",
3      method = RequestMethod.GET,
4      produces = MediaType.APPLICATION_JSON_VALUE)
5  public ResponseEntity<Issue> getIssue(@PathVariable("id") Long id) {
6      logger.info("> getIssue");
7
8      Issue issue = null;
9      try {
10         issue = issueService.find(id);
11
12         // if no issue found, return 404 status code
13         if (issue == null) {
14             return new ResponseEntity<Issue>(HttpStatus.NOT_FOUND);
```

```
15         }
16     } catch (Exception e) {
17         logger.error("Unexpected Exception caught.", e);
18         return new ResponseEntity<Issue>(issue,
19             HttpStatus.INTERNAL_SERVER_ERROR);
20     }
21
22     logger.info("< getIssue");
23     return new ResponseEntity<Issue>(issue, HttpStatus.OK);
24 }
```

The context path to find a single entity is slightly different than what is used to fetch a collection. Rather than just the plural entity name, `/issues`, the path also contains the primary key identifier of the entity instance to be retrieved. In the Spring Framework, context path variables are delimited by curly braces. The value of the `id` context path variable may optionally be passed in a method parameter annotated with `@PathVariable`.

The HTTP method used to fetch entities is `GET`. Therefore, to retrieve an issue entity whose primary key value is 1, a HTTP `GET` request is sent to `/issues/1`.

The request mapping annotation contains a `produces` element whose value informs the Spring Framework that requests must have an `Accepts` HTTP header value of `application/json`. The Spring Framework also uses the `produces` element value to convert the response entity into the appropriate form before placing it into the HTTP response body. In our case, Spring converts the `Issue` object into the equivalent JSON representation.

The controller method invokes the `find` method on the `issueService` passing the primary key identifier obtained from the web service context path. If the service is unable to find a matching `Issue` in the database, the web service returns HTTP status code 404 to indicate the requested resource could not be found. If the service returns an `Issue` object, the controller returns the object and HTTP status code 200 to indicate the request completed successfully.

User Interface

With the `getIssue` web service completed, the user interface may be enhanced to retrieve individual issues and display them to the user. To do this, we will create a HTML template that shows all of the issue attributes and a new `ItemView` class to manage user interaction with the template. The list item template will be updated to trigger a new event when users click an issue title. The issue manager controller will be modified to orchestrate the interaction between the views.

Entities Module

Issue models are retrieved from the server in a process similar to that which is used to retrieve collections of issues. So that the Issue Manager module may request specific, individual issue models

from the server, we must update the `Entities` module to handle requests for a single `Issue` using the `id` attribute.

Controller

The `IssueEntityController` class contains functions that request issue information from the server. In the `src/main/app/js/modules/entities` directory, open the `Issue.js` source file. Within the `IssueEntityController` class, create a new function named `getIssue` as depicted below.

Issue Entity Controller - Get Issue

```
1  getIssue: function(issueId) {
2    logger.debug("IssueEntityController.getIssue");
3    var issue = new Entities.Issue({ id: issueId });
4    var defer = $.Deferred();
5    issue.fetch({
6      success: function(data) {
7        defer.resolve(data);
8      }
9    });
10   return defer.promise();
11 },
```

The function receives a single parameter, `issueId`, which contains the primary key identifier value of the issue entity to be retrieved from the server. A new `Issue` model is created, initializing the `id` attribute with the primary key value.

A jQuery `Deferred` object is created and its `Promise` is returned from the function. Returning the `Promise` allows the calling logic to supply `done` and `fail` functions to be invoked based upon the server response.

The model's `fetch` function is called. Since the model's `id` attribute is populated, the `Backbone.sync` function appends the `id` attribute value to the model's `urlRoot` and sends an HTTP GET request to the service at that location. The success handler function calls the `resolve` function on the `Deferred` object passing the data returned by the web service. In turn, any success function callbacks registered on the `Promise` returned by this function will be called in the order in which they were added to the `Promise`.

Later in this chapter, we will use the `Promise` object returned by this function within the `Issue Manager` module controller.

Request Response Handler

So that any application component may retrieve individual issue models, a request/response message handler function is defined in the `Entities` module. In the `src/main/app/js/modules/entities` directory, open the `Issue.js` source file. Above the request handler function for `Issue` collections, add the following logic.

Issue Entity Request Handler

```
1 // Handle request for an Issue Model
2 IssueTrackerApp.reqres.setHandler('issue:entity', function(id) {
3   logger.debug("Handling 'issue:entity' request");
4   return issueController.getIssue(id);
5 });
```

The request message is named `issue:entity`. The primary key identifier is passed as a parameter by the sender. The request handler function executes the `getIssue` function on the `IssueEntityController` and returns the Promise object.

Issue Manager Module

The issue manager module encapsulates all activities related to issues. To implement the ability to view individual issues, we need to modify the list item template to create a link on issue titles and the list item view to handle click events on that link. We will also enhance the controller to orchestrate the behaviors necessary to transition from the list view to show the details of a single issue.

View Issue HTML Template

The HTML template to display individual issue entities is divided into three sections: a heading, an actions bar, and the issue attributes. When the template is rendered, the issue attributes are evaluated to dynamically render the HTML markup.

In the `src/main/app/templates` directory, create a new file named `issueview.html` and place the following HTML markup within it.

View Issue HTML Template

```
1 <div class="row">
2   <div class="col-md-12">
3     <div class="page-header">
4       <h1><%= title %></h1>
5     </div>
6   </div>
7 </div>
8
9 <div class="row"> <!-- begin actions row -->
10   <div class="col-md-12"> <!-- begin actions column -->
11
12     <div class="actionbar">
13       <button type="button" class="btn btn-default js-list">
```

```

14         <i class="fa fa-list fa-fw"></i>
15         List
16     </button>
17 </div>
18
19 </div> <!-- end actions column -->
20 </div> <!-- end actions row -->
21
22 <div class="row"> <!-- begin details row -->
23     <div class="col-md-12"> <!-- begin details column -->
24
25         <h4>Details:</h4>
26         <div class="panel panel-default">
27             <div class="panel-body">
28
29                 <div class="row"> <!-- begin type row -->
30
31                     <div class="col-md-2"> <!-- begin 1st column -->
32                         Type:
33                     </div> <!-- end 1st column -->
34
35                     <div class="col-md-4"> <!-- begin 2nd column -->
36                         <% switch( type ) {
37                             case "TASK": { %>
38                             <span>
39                                 <i class="fa fa-check fa-fw" title="Task"></i>
40                                 Task
41                             </span>
42                             <% break;
43                             }
44                             case "STORY": { %>
45                             <span>
46                                 <i class="fa fa-book fa-fw text-primary" title="Story"></i>
47                                 Story
48                             </span>
49                             <% break;
50                             }
51                             case "ENHANCEMENT": { %>
52                             <span>
53                                 <i class="fa fa-wrench fa-fw text-muted" title="Enhancement"></i>
54                                 Enhancement
55                             </span>

```

```

56         <% break;
57     }
58     case "BUG": { %>
59         <span>
60             <i class="fa fa-bug fa-fw text-danger" title="Bug"></i>
61             Bug
62         </span>
63         <% break;
64     }
65     } %>
66 </div> <!-- end 2nd column -->
67
68 </div> <!-- end type row -->
69 <div class="row"> <!-- begin priority row -->
70
71     <div class="col-md-2"> <!-- begin 1st column -->
72         Priority:
73     </div> <!-- end 1st column -->
74
75     <div class="col-md-4"> <!-- begin 2nd column -->
76     <% switch( priority ) {
77         case "LOW": { %>
78         <span>
79             <i class="fa fa-arrow-down fa-fw text-success" title="Low"></i>
80             Low
81         </span>
82         <% break;
83     }
84     case "MEDIUM": { %>
85     <span>
86         <i class="fa fa-chevron-up fa-fw text-primary" title="Medium"></i>
87         Medium
88     </span>
89     <% break;
90 }
91 case "HIGH": { %>
92 <span>
93     <i class="fa fa-arrow-up fa-fw text-danger" title="High"></i>
94     High
95 </span>
96 <% break;
97     }

```

```

98         } %>
99     </div> <!-- end 2nd column -->
100
101 </div> <!-- end priority row -->
102 <div class="row"> <!-- begin status row -->
103
104     <div class="col-md-2"> <!-- begin 1st column -->
105         Status:
106     </div> <!-- end 1st column -->
107
108     <div class="col-md-4"> <!-- begin 2nd column -->
109     <% switch( status ) {
110         case "OPEN": { %>
111         <span class="label label-primary">Open</span>
112         <% break;
113         }
114         case "IN_PROGRESS": { %>
115         <span class="label label-warning">In Progress</span>
116         <% break;
117         }
118         case "DONE": { %>
119         <span class="label label-success">Done</span>
120         <% break;
121         }
122     } %>
123     </div> <!-- end 2nd column -->
124
125 </div> <!-- end status row -->
126
127 </div>
128 </div>
129
130 </div> <!-- end details column -->
131 </div> <!-- end details row -->
132
133 <div class="row"> <!-- begin description row -->
134     <div class="col-md-12"> <!-- begin description column -->
135
136     <h4>Description:</h4>
137     <pre class="panel-pre"><%= description %></pre>
138
139 </div> <!-- end description column -->

```

```
140 </div> <!-- end description row -->
```

The template contains several, full-width rows. The first displays the page heading which is the issue's title.

The second row contains an action bar, which is simply a row of buttons to perform actions on the issue currently displayed or to navigate to another section of the application. At this time, the action bar contains a single button used to return the user to the list of issues. In the next chapter, we will add a button allowing users to edit the issue.

The third row contains the HTML markup to display the `type`, `priority`, and `status` attribute values.

The final row contains the value of the `description` attribute. The description is placed inside a `pre` tag to *preserve* the control characters, such as tabs and carriage controls.

Custom CSS styles are required to create the desired appearance for the action bar and issue description.

CSS

The `itemview.html` template created in the previous section uses two new custom CSS classes which must be added to the application stylesheet. In the `src/main/app/css` directory, open the `app.css` source file and add the `actionbar` rule set.

actionbar CSS Rule Set

```
1 .actionbar {  
2     margin-bottom: 20px;  
3 }
```

This simple rule set creates 20 pixels of margin beneath the action bar row on the issue view HTML template. Without the extra spacing, the action bar buttons would appear uncomfortably close to the content immediately below them.

Next, add the `panel-pre` CSS rule set depicted below.

panel-pre CSS Rule Set

```
1 .panel-pre {  
2     padding: 15px;  
3     margin-bottom: 20px;  
4     background-color: #fff;  
5     font-family: "Helvetica Neue", Helvetica, Arial, sans-serif;  
6     font-size: 14px;  
7 }
```

The Bootstrap framework has a panel component. A Bootstrap panel creates a box with rounded corners surrounding HTML content. Panels may optionally have headers and footers and offer specializations for tables and list groups. The Bootstrap panel CSS rule sets are named `panel-*`. For example, `panel-primary` styles a panel with the primary color scheme.

We want to utilize the HTML `pre` tag's native browser behavior, but style it to appear like a Bootstrap panel. The `panel-pre` rule set creates a font, color, and spacing scheme that mimics a Bootstrap panel.

View Issue ItemView

Let's create a view class to render the HTML template and manage user interaction with the action bar. Since the view represents a single issue entity model rather than a collection, create a class extending `ItemView` to manage the `issueview.html` template.

Open the `IssueManagerViews.js` source file located in the `issuemanager` module directory. At the bottom of the module definition callback method, create the `IssueView` class illustrated below.

IssueView

```
1 // Define the View for a Single Issue  
2 IssueManager.IssueView = Backbone.Marionette.ItemView.extend({  
3  
4     className: 'container-fluid',  
5  
6     template: 'issueview',  
7  
8     triggers: {  
9         'click .js-list': 'issue:list'  
10    }  
11  
12 });
```

The `tagName` attribute is omitted and, therefore, the base HTML DOM element defaults to `div`. The `className` attribute applies the `container-fluid` class to the `div` tag.

The `template` attribute value assigns the `issueview.html` HTML template to this view class.

The template contains an action bar with a single *List* button. The `triggers` hash maps click events on the *List* button to an event named `issue:list`. When a user clicks the *List* button, the view publishes an event named `issue:list`. Later in this chapter, we will create controller logic to listen for this event and take action when it occurs.

List Item

We need to create a way for users to select an issue from the list and access the new issue view. Let's update each list item to make the issue title a hyperlink to view the issue detail.

List Item HTML Template

Edit the `issuelistitem.html` template. Update the column that displays the title attribute as illustrated below.

List Item HTML Template

```
1 <td class="col-md-9">
2   <a href="#" class="js-view"><%= title %></a>
3 </td>
```

Place the underscore template markup within a HTML anchor tag. The CSS class `js-view` will be used by the list item view to listen for click events on the DOM element.

The `href` attribute is not needed; however, it is a required attribute on anchor tags. Set the value of the `href` attribute to `#`, which is an empty hash route. The application logic that listens for click events on the anchor tag prevents execution of the default browser behavior invoked when an anchor tag is clicked. Therefore, the value in the `href` attribute is irrelevant in this case.

List Item View Class

Open the `IssueManagerViews.js` source file and locate the `IssueListItemView` class. Let's create a `triggers` hash to listen for click events on the issue title.

IssueListItemView

```
1  // Define the View for a single Issue in the List
2  IssueManager.IssueListItemView = Backbone.Marionette.ItemView.extend({
3
4      tagName: 'tr',
5
6      template: 'issuelistitem',
7
8      triggers: {
9          'click .js-view': 'issue:view'
10     }
11
12 });
```

The complete view class is illustrated above with the new `triggers` hash highlighted.

The hash maps DOM click events on elements with the CSS class `js-view` to an event named `issue:view`. Put simply, when a user clicks the anchor element containing the issue title, the `IssueListItemView` will publish an event named `issue:view`.

Issue Manager Controller

We created a new template and view class to display the details of a single issue entity. We enhanced the list item template and view so that a user can select an issue from the list to view. Let's modify the issue manager controller to handle the events triggered by these views, transitioning the views from the list to a single issue and vice versa.

Controller View Function

We need to add a function to the `IssueManagerController` to create an instance of `IssueView` and show it in the main application region. The controller also listens for `issue:list` events triggered by the view, returning the user to the list of issues.

Open the `IssueController.js` source file. Below the `add` function, define a new function named `view` as illustrated below.

IssueManagerController - view Function

```
1  view: function(id, model, collection) {
2      logger.debug("IssueManagerController.view id:" + id);
3
4      var displayView = function(issueModel, issueCollection) {
5          var issueView = new IssueManager.IssueView({
6              model: issueModel
7          });
8
9          issueView.on('issue:list', function(args) {
10             logger.debug("Handling 'issue:list' event");
11             IssueTrackerApp.execute('issuemanager:list', issueCollection);
12         });
13
14         logger.debug("Show IssueView in IssueTrackerApp.mainRegion");
15         IssueTrackerApp.mainRegion.show(issueView);
16     };
17
18     if(model) {
19         displayView(model, collection);
20     } else {
21         var fetchingIssue = IssueTrackerApp.request('issue:entity', id);
22         $.when(fetchingIssue).done(function(issue) {
23             displayView(issue, collection);
24         });
25     }
26 }
```

The view function accepts three parameters: the primary key identifier, an issue model, and a collection of issues. Only the primary key identifier is required so that an issue model may be fetched from the server. The model and collection parameters, when available, are used to prevent unnecessary web service requests since the application already has the entities needed.

The view function defines a private function named `displayView` which performs the view instantiation and event handler logic. The `displayView` function is passed an issue model to be shown and an optional collection of issues if the view operation was triggered from within the list view.

When the `issue:list` event is triggered by the view, the controller transitions back to the issue list.

After the `displayView` function definition, the controller analyzes the parameters passed to the view function. If the model parameter is undefined, the model is fetched from the server using the primary key identifier and the `displayView` function is called with the fetched model. However, if an issue model was passed to the view function, that model instance is passed to `displayView`.

View Issue Command Handler

Create a command handler function subscribing to `issuemanager:view` commands. Within the `IssueManagerController.js` source file, below the command handler for adding issues, insert the following logic.

View Issue Command Handler

```
1   IssueTrackerApp.commands.setHandler('issuemanager:view', function(id, model, c\
2 collection) {
3     logger.debug("Handling 'issuemanager:view' command");
4     IssueTrackerApp.navigate('issues/' + id);
5     controller.view(id, model, collection);
6   });
```

Upon receipt of an `issuemanager:view` command, the handler function updates the browser's history and address bar by calling the `navigate` function. Next, the controller's `view` function is invoked passing the `id`, `model`, and `collection` parameters.

List Function Event Handler

The request to view an issue is initiated within the list of issues. Let's modify the issue manager controller's `list` function to handle requests to view issues.

Open the `IssueManagerController.js` source file and update the `list` function of `IssueManagerController` as shown below.

IssueManagerController - list Function

```
1   list: function(collection) {
2     logger.debug("IssueManagerController.list");
3
4     var displayListView = function(issueCollection) {
5       var listView = new IssueManager.IssueListView({
6         collection: issueCollection
7       });
8
9       // Handle 'issue:view' events triggered by Child Views
10      listView.on('childview:issue:view', function(args) {
11        logger.debug("Handling 'childview:issue:view' event");
12        IssueTrackerApp.execute('issuemanager:view', args.model.get('id'), arg\
13 s.model, issueCollection);
14      });
15
```

```
16     logger.debug("Show IssueListView in IssueTrackerApp.mainRegion");
17     IssueTrackerApp.mainRegion.show(listView);
18 };
19
20     if(collection) {
21         displayListView(collection);
22     } else {
23         var fetchingIssues = IssueTrackerApp.request('issue:entities');
24         $.when(fetchingIssues).done(function(issues) {
25             displayListView(issues);
26         });
27     }
28 },
```

The complete list function is depicted above with the new logic highlighted.

Add an event listener to `listView` for the `issue:view` event triggered by `IssueListItemView`. `CompositeViews` have a special relationship with their child views. Events triggered by child views are republished by the `CompositeView` with `childview:` prepended to the event name. Therefore, in the controller, we can handle events triggered by the `IssueListItemView` by listening to the `IssueListView`.

When the `childview:issue:view` event is received, the controller executes an application command named `issuemanager:view`. Trigger event handler functions receive a single argument with three properties: `view`, `model`, and `collection`. These properties are the triggering view, the view's model, and the view's collection. The controller passes the issue identifier, the issue model, and the collection of issues as command parameters. Therefore, when accessing the issue view via the list of issues, the model and collection are passed to the view controller function, eliminating the need to fetch the single issue from the web services.

Issue Manager Router

The view issue command handler updates the browser history and address bar with a hash route containing the primary key identifier of the issue whose detail is displayed. If a user bookmarks this route, clicks the browser back button, or enters a URL in this format into the address bar, the Marionette application must handle this route and display the corresponding issue's detail.

Let's add a route to the `IssueManagerRouter` that maps the URL pattern for viewing a single issue to the controller view function. In the `IssueManagerController.js` source file, update the `IssueManagerRouter` as illustrated below.

IssueManagerRouter

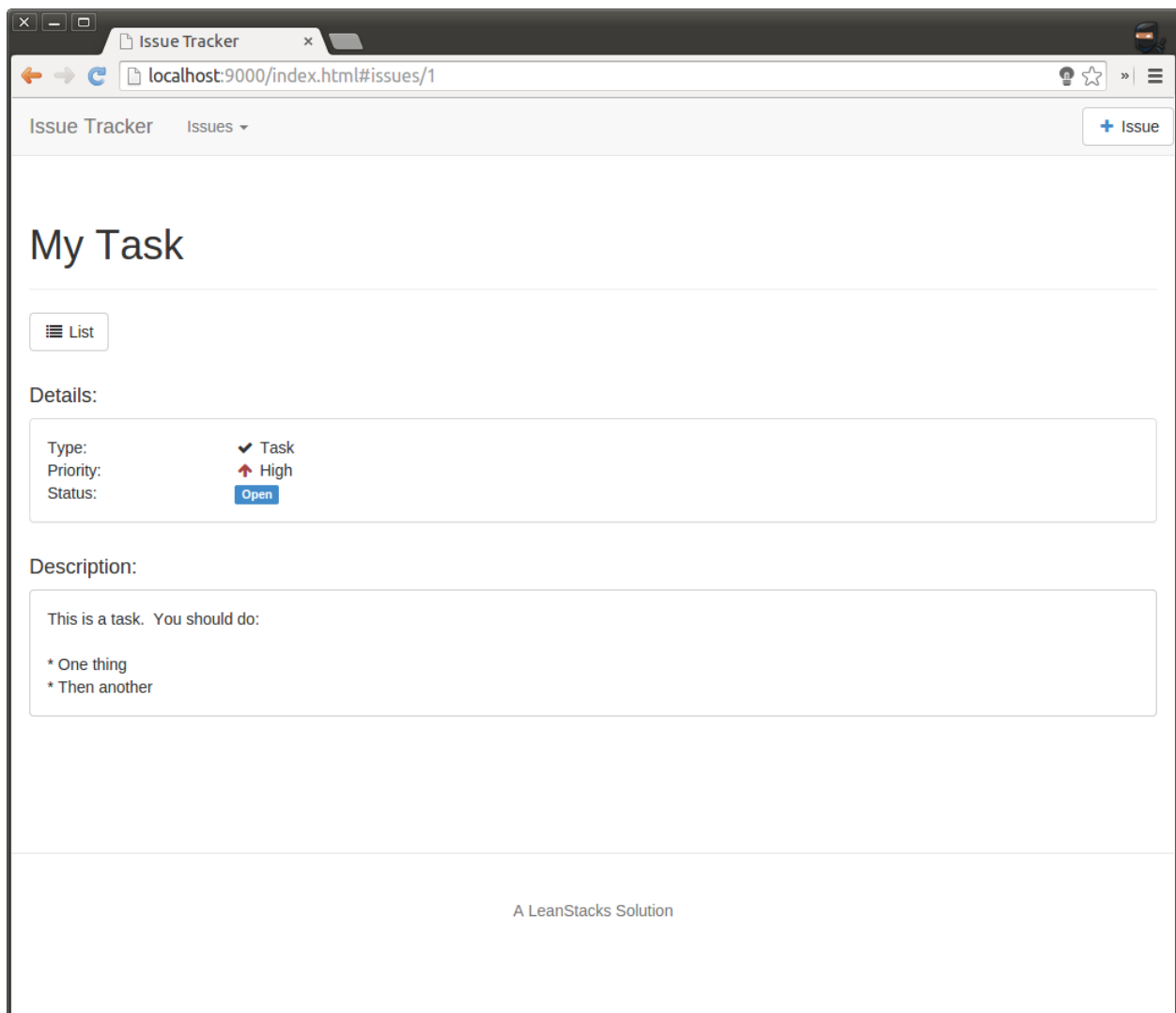
```
1  // Define the AppRouter for the IssueManager module
2  var IssueManagerRouter = Marionette.AppRouter.extend({
3
4      appRoutes: {
5          'issues': 'list',
6          'issues/:id': 'view'
7      }
8
9  });
```

The new route is highlighted in the `appRoutes` hash. The route `issues/:id` contains a path variable, `:id`. The router extracts the value from this segment of the path and passes it as a parameter to the mapped controller function. For example, entering the route `#issues/1` in the browser address bar results in calling `view(1)` on the `IssueManagerController`. Notice that the model and collection parameters are not passed to the controller `view` function when called by the `IssueManagerRouter`. Therefore, the issue model is fetched from the web service when the `view` function is called from the router.

Running the Application

Let's test the new functionality. Open a terminal window and navigate to the web services project base directory. Type `mvn spring-boot:run` and press *Enter* to start the web services on port 8080. Open a second terminal window, or tab, and navigate to the user interface project base directory. Type `grunt run` and press *Enter* to start the local web server on port 9000.

Open a browser and press `F12` to open the developer tools. In the browser's address bar type `http://localhost:9000/index.html` and press *Enter*. The issue manager module displays an empty issue list. Click the add issue button in the top navigation bar and create a new issue. Click the title of the issue in the list to show the issue detail. If everything works correctly, the user interface displays the issue details. Click the *List* button to return to the list of issues.



Issue Manager - View

In the console tab of the developer tools, log statements depict the application flow.

Console Log

```
1 09:38:17 DEBUG - IssueTrackerApp.start
2 09:38:17 DEBUG - Footer initializer
3 09:38:17 DEBUG - FooterController.show
4 09:38:17 DEBUG - Header initializer
5 09:38:17 DEBUG - HeaderController.show
6 09:38:17 DEBUG - IssueManager initializer
7 09:38:17 DEBUG - Backbone.history.start
8 09:38:17 DEBUG - Handling 'issuemanager:list' command
9 09:38:17 DEBUG - IssueTrackerApp.navigate route:issues
```

```
10 09:38:17 DEBUG - IssueManagerController.list
11 09:38:17 DEBUG - Handling 'issue:entities' request
12 09:38:17 DEBUG - IssueEntityController.getIssues
13 09:38:17 DEBUG - Show IssueListView in IssueTrackerApp.mainRegion
14 09:38:21 DEBUG - Handling 'issuemanager:add' command
15 09:38:21 DEBUG - IssueManagerController.add
16 09:38:21 DEBUG - Show IssueAddView in IssueTrackerApp.mainRegion
17 09:38:53 DEBUG - IssueAddView.onCreateClicked
18 09:38:53 DEBUG - Handling 'form:submit' event
19 09:38:53 DEBUG - form data:{"type":"TASK","priority":"HIGH","title":"My Task","description":"This is a task. You should do:\n\n* One thing\n* Then another"}
20
21 09:38:53 DEBUG - Handling 'issuemanager:view' command
22 09:38:53 DEBUG - IssueTrackerApp.navigate route:issues/1
23 09:38:53 DEBUG - IssueManagerController.view id:1
24 09:38:53 DEBUG - Show IssueView in IssueTrackerApp.mainRegion
25 09:39:30 DEBUG - Handling 'issue:list' event
26 09:39:30 DEBUG - Handling 'issuemanager:list' command
27 09:39:30 DEBUG - IssueTrackerApp.navigate route:issues
28 09:39:30 DEBUG - IssueManagerController.list
29 09:39:30 DEBUG - Handling 'issue:entities' request
30 09:39:30 DEBUG - IssueEntityController.getIssues
31 09:39:30 DEBUG - Show IssueListView in IssueTrackerApp.mainRegion
32 09:39:33 DEBUG - Handling 'childview:issue:view' event
33 09:39:33 DEBUG - Handling 'issuemanager:view' command
34 09:39:33 DEBUG - IssueTrackerApp.navigate route:issues/1
35 09:39:33 DEBUG - IssueManagerController.view id:1
36 09:39:33 DEBUG - Show IssueView in IssueTrackerApp.mainRegion
```

Lines 1 through 13 depict the Marionette application startup flow. Lines 14 through 20 are logged as the new issue is created. Line 21 is logged after the new issue is created, triggering the display of that individual issue. The IssueView instance is shown in the main region of the application. Line 25 indicates the *List* button was clicked. Lines 25 through 31 are logged as the application transitions to the issue list. Finally, lines 32 through 36 are logged when the issue title is clicked in the list and the application displays the issue detail view again.



The Road Ahead

In the next chapter, we will enhance both the web services and user interface to update issues.

Updating Issues

The web services and user interface are able to create, list, and view issues. What if an issue needs to be altered after it is created? Perhaps the priority changes. Perhaps new information is available that clarifies the description. These scenarios require the ability to update issue attributes.

In this chapter, we will create a new web service that updates an issue previously created and persisted in the database. To provide the ability to modify issue attributes, we will enhance the user interface to facilitate editing issues in the list.

Web Service

Constructing a web service to update issue entities requires changes to the business service interface, the service implementation bean, and the web service controller. The logic is similar to that used to create new issues.

Service Interface

The service interface defines the contract for the service implementation class. Previously the create method was declared in the `IssueService` interface. Just below it, let's define a new method to update Issue entities.

Issue Service - Update

```
1  /**
2   * Update an Issue entity in the data repository.
3   * @param issue An issue entity to update.
4   * @return The updated issue entity.
5   */
6  Issue update(Issue issue);
```

The update method receives a modified Issue entity whose state changes need to be stored in the database. The method returns the updated, persisted entity.

Service Implementation

The service implementation class implements the functionality for methods declared on the service interface. We need to add the logic for the update method that we declared on the `IssueService` interface. Open the `IssueServiceBean` add the following code snippet below the create method.

Issue Service Implementation - Update

```
1  @Override
2  public Issue update(Issue issue) {
3      logger.info("> update");
4
5      Issue updatedIssue = issueRepository.save(issue);
6
7      logger.info("< update");
8      return updatedIssue;
9  }
```

The Spring Data framework implements all of the logic to connect to the database, generate SQL update statements, and store the contents of the Issue entity in the database. The save method returns the persisted version of the Issue entity. The updated, persisted Issue entity is returned to the service client.



Repository save Method

The save method is used to both create and update Issue entities in the database. Spring Data examines the primary key identifier attribute annotated with `@Id` to determine if the entity has previously been created. If the `@Id` attribute contains a value, Spring Data will generate a SQL UPDATE statement. If the attribute is null, an INSERT statement is generated.

Controller

The controller class implements the RESTful web service request handler methods. Let's code the request handler method and annotations that will receive requests to update Issue entities. Open the `IssueController` class and add the following logic below the `createIssue` request handler method.

Issue Controller - Update

```
1  @RequestMapping(
2      value = "/issues/{id}",
3      method = RequestMethod.PUT,
4      consumes = MediaType.APPLICATION_JSON_VALUE,
5      produces = MediaType.APPLICATION_JSON_VALUE)
6  public ResponseEntity<Issue> updateIssue(@RequestBody Issue issue) {
7      logger.info("> updateIssue");
8
9      Issue updatedIssue = null;
10     try {
```



```
11         updatedIssue = issueService.update(issue);
12     } catch (Exception e) {
13         logger.error("Unexpected Exception caught.", e);
14         return new ResponseEntity<Issue>(HttpStatus.INTERNAL_SERVER_ERROR);
15     }
16
17     logger.info("< updateIssue");
18     return new ResponseEntity<Issue>(updatedIssue, HttpStatus.OK);
19 }
```

The context path to update entities is slightly different than what is used create them. Rather than just the plural entity name, `/issues`, the path also contains the primary key identifier of the entity instance to be updated. In the Spring Framework context path variables are delimited by curly braces. The value of the `id` context path variable may optionally be passed in a method parameter annotated with `@PathVariable`; however, the `id` path variable is not needed here, so the method parameter is omitted.

The HTTP method used to update entities is `PUT`. Therefore, to update an issue entity whose primary key value is 1, a HTTP `PUT` request is sent to `/issues/1`.

Like the request handler to create issues, the update method is annotated to declare that this method *consumes* JSON, telling the Spring Framework to map this method to requests whose `Content-Type` HTTP header value is `application/json`. Similarly, the method *produces* JSON, telling the Spring Framework that requests must have an `Accepts` HTTP header value of `application/json`.

The `@RequestBody` annotation is applied to the `issue` method parameter. Spring automatically uses the appropriate library to marshal the contents of the HTTP request body into the annotated parameter. The primary key identifier is included in the JSON request body in addition to the context path. The `IssueService` uses the primary key identifier marshaled from the request body to update the issue.

User Interface

With the `updateIssue` web service in place, the user interface may be enhanced to edit issue entities and send update requests to the server. To do this, we will create a HTML template with the form to edit an issue and a new `ItemView` class to manage user interaction with the form. The list and list item templates will be updated to create an *Edit* button for each issue in the list. The issue manager controller will be modified to orchestrate the interaction between the views.

A new column named *Actions* will be created in the list of issues. This column will contain buttons for actions on an individual issue. When the *Edit* button is clicked, the application will show the edit view. When the issue is updated, Marionette automatically updates the attributes of the issue model in the collection displayed by the list view and the controller transitions to the single issue view.

Edit Issue HTML Template

The HTML template to update issues is similar to the template used to create new issues with a few small differences. When creating an issue, the `status` attribute is defaulted to *Open*; however, the user should be able to update the `status` value when editing an issue. We need to add a button group to the form allowing the user to modify the value of the `status`.

When the template is initially rendered, the form needs to be populated with the attribute values of the issue being edited.

In the `src/main/app/templates` directory, create a new file named `issueedit.html` and place the following HTML markup within it.

Edit Issue HTML Template

```

1  <div class="row">
2    <div class="col-md-12">
3      <div class="page-header">
4        <h1>Edit Issue</h1>
5      </div>
6    </div>
7  </div>
8
9  <div class="row"> <!-- begin outer row -->
10    <div class="col-md-12"> <!-- begin outer column -->
11
12      <form role="form">
13
14        <div class="row"> <!-- begin nested row -->
15
16          <div class="col-md-4"> <!-- begin 1st column -->
17
18            <div class="form-group">
19              <div class="btn-group btn-group-sm" data-toggle="buttons">
20                <label class="btn btn-default <% if( type === 'TASK') { %> active \
21 <% } %>">
22                  <input type="radio"
23                    name="type"
24                    value="TASK"
25                    <% if( type === 'TASK') { %> checked <% } %>
26                    autocomplete="off">
27                  <i class="fa fa-check fa-fw"></i> Task
28                </input>
29              </label>
30              <label class="btn btn-default <% if( type === 'STORY') { %> active \

```

```

31  <% } %>">
32      <input type="radio"
33          name="type"
34          value="STORY"
35          <% if( type === 'STORY') { %> checked <% } %>
36          autocomplete="off">
37      <i class="fa fa-book fa-fw text-primary"></i> Story
38  </input>
39  </label>
40  <label class="btn btn-default <% if( type === 'ENHANCEMENT') { %> \
41  active <% } %>">
42      <input type="radio"
43          name="type"
44          value="ENHANCEMENT"
45          <% if( type === 'ENHANCEMENT') { %> checked <% } %>
46          autocomplete="off">
47      <i class="fa fa-wrench fa-fw text-muted"></i> Enhancement
48  </input>
49  </label>
50  <label class="btn btn-default <% if( type === 'BUG') { %> active <\
51  % } %>">
52      <input type="radio"
53          name="type"
54          value="BUG"
55          <% if( type === 'BUG') { %> checked <% } %>
56          autocomplete="off">
57      <i class="fa fa-bug fa-fw text-danger"></i> Bug
58  </input>
59  </label>
60  </div>
61  </div>
62
63  </div> <!-- end 1st column -->
64
65  <div class="col-md-3"> <!-- begin 2nd column -->
66
67      <div class="form-group">
68          <div class="btn-group btn-group-sm" data-toggle="buttons">
69              <label class="btn btn-default <% if( priority === 'LOW') { %> acti\
70  ve <% } %>">
71                  <input type="radio"
72                      name="priority"

```

```

73         value="LOW"
74         <% if( priority === 'LOW') { %> checked <% } %>
75         autocomplete="off">
76         <i class="fa fa-arrow-down fa-fw text-success"></i> Low
77     </input>
78 </label>
79 <label class="btn btn-default <% if( priority === 'MEDIUM') { %> a\
80 ctive <% } %>">
81     <input type="radio"
82         name="priority"
83         value="MEDIUM"
84         <% if( priority === 'MEDIUM') { %> checked <% } %>
85         autocomplete="off">
86     <i class="fa fa-chevron-up fa-fw text-primary"></i> Medium
87 </input>
88 </label>
89 <label class="btn btn-default <% if( priority === 'HIGH') { %> act\
90 ive <% } %>">
91     <input type="radio"
92         name="priority"
93         value="HIGH"
94         <% if( priority === 'HIGH') { %> checked <% } %>
95         autocomplete="off">
96     <i class="fa fa-arrow-up fa-fw text-danger"></i> High
97 </input>
98 </label>
99 </div>
100 </div>
101
102 </div> <!-- end 2nd column -->
103
104 <div class="col-md-3"> <!-- begin 3rd column -->
105
106     <div class="form-group">
107         <div class="btn-group btn-group-sm" data-toggle="buttons">
108             <label class="btn btn-default <% if( status === 'OPEN') { %> activ\
109 e <% } %>">
110                 <input type="radio"
111                     name="status"
112                     value="OPEN"
113                     <% if( status === 'OPEN') { %> checked <% } %>
114                     autocomplete="off">

```

```

115         Open
116         </input>
117     </label>
118     <label class="btn btn-default <% if( status === 'IN_PROGRESS') { %\
119 > active <% } %>">
120         <input type="radio"
121             name="status"
122             value="IN_PROGRESS"
123             <% if( status === 'IN_PROGRESS') { %> checked <% } %>
124             autocomplete="off">
125         In Progress
126     </input>
127 </label>
128 <label class="btn btn-default <% if( status === 'DONE') { %> activ\
129 e <% } %>">
130     <input type="radio"
131         name="status"
132         value="DONE"
133         <% if( status === 'DONE') { %> checked <% } %>
134         autocomplete="off">
135     Done
136 </input>
137 </label>
138 </div>
139 </div>
140
141 </div> <!-- end 3rd column -->
142
143 </div> <!-- end nested row -->
144
145 <div class="form-group">
146     <label for="issue-title-input"
147         class="control-label">
148         Title
149     </label>
150     <input id="issue-title-input"
151         type="text"
152         name="title"
153         class="form-control"
154         placeholder="Enter a brief title..."
155         autofocus
156         value="<%= title %>" />

```

```

157     </div>
158
159     <div class="form-group">
160         <label for="issue-description-textarea"
161             class="control-label">
162             Description
163         </label>
164         <textarea id="issue-description-textarea"
165             name="description"
166             class="form-control"
167             rows="6"
168             placeholder="Describe the issue..."><%= description %></textare\
169 ea>
170     </div>
171
172     <button type="button"
173         class="btn btn-primary js-update"
174         data-loading-text="Updating...">
175         Update
176     </button>
177     <button type="button"
178         class="btn btn-link js-cancel">
179         Cancel
180     </button>
181
182 </form>
183
184 </div> <!-- end outer column -->
185 </div> <!-- end outer row -->

```

The template contains two, full-width rows. The first displays the *Edit Issue* heading above the form. The second row contains the form. Since the HTML markup is similar to the create issue template, let's focus on the differences.

The button groups that display the selected value of the type, priority, and status attributes require a CSS class named `active` on the `label` tag of the selected value. Within the template, an `if` statement is added within the `css` attribute of each `label` tag, adding the `active` class when appropriate. Similarly, the input element requires a `checked` attribute to indicate that it is the selected button within the group. Another `if` statement is used to add the `checked` attribute to the input element whose value matches the issue being edited.

The issue title is displayed in a standard `<input type="text" ... />` form tag. The title is set in the `value` attribute of the input tag.

The issue description is displayed in a `textarea` tag. Textarea tags are used to gather multi-line user input. Textarea tags do not use the `value` attribute. Instead, the value of a textarea is placed between the opening and closing tags. Any whitespace contained between the textarea tags is included in the value of the form field.

The button labeled *Update* triggers the form submission and the button labeled *Cancel* returns to the issue view. Note the familiar `js-` prefixed CSS classes on the buttons that will be used as UI element selectors in the `ItemView` class managing this template.

Edit Issue View

Let's create a view class to render the HTML template and manage user interaction with the edit issue form. Since the view works with a single issue entity model rather than a collection, create a class extending `ItemView`.

Open the `IssueManagerViews.js` source file located in the `issuemanager` module directory. At the bottom of the module definition callback function, create the `IssueEditView` class illustrated below.

IssueEditView

```
1  // Define the View for Editing an Issue
2  IssueManager.IssueEditView = Backbone.Marionette.ItemView.extend({
3
4      className: 'container-fluid',
5
6      template: 'issueedit',
7
8      ui: {
9          updateButton: 'button.js-update',
10         cancelButton: 'button.js-cancel'
11     },
12
13     events: {
14         'click @ui.updateButton': 'onUpdateClicked',
15     },
16
17     triggers: {
18         'click @ui.cancelButton': 'form:cancel'
19     },
20
21     onUpdateClicked: function(e) {
22         logger.debug("IssueEditView.onUpdateClicked");
23         e.preventDefault();
24         this.showProcessingState();
```

```

25     var data = Backbone.Syphon.serialize(this);
26     this.trigger('form:submit', data);
27 },
28
29 showProcessingState: function() {
30     var spinnerContent = '<i class="fa fa-circle-o-notch fa-spin"></i> ';
31     this.ui.updateButton.button('loading');
32     this.ui.updateButton.prepend(spinnerContent);
33 },
34
35 hideProcessingState: function() {
36     this.ui.updateButton.button('reset');
37 },
38
39 onFormValidationFailed: function(errors) {
40     this.hideProcessingState();
41     this.hideFormErrors();
42     _.each(errors, this.showFormError, this);
43 },
44
45 showFormError: function(errorMessage, fieldKey) {
46     var $formControl = this.$el.find('[name="'+fieldKey+'"]');
47     var $controlGroup = $formControl.parents('.form-group');
48     var errorContent = '<span class="help-block js-form-error">'+errorMessage+\'
49 </span>';
50     $formControl.after(errorContent);
51     $controlGroup.addClass('has-error');
52 },
53
54 hideFormErrors: function() {
55     this.$el.find('.js-form-error').each(function() {
56         $(this).remove();
57     });
58     this.$el.find('.form-group.has-error').each(function() {
59         $(this).removeClass('has-error');
60     });
61 }
62
63 });

```

This class is very similar to the `IssueAddView` class created previously. The button to submit the form is mapped to the UI element named `updateButton` rather than `createButton`. Otherwise the

view functionality mirrors that of the `IssueAddView`. The shared logic from the `IssueAddView` and `IssueEditView` could be extracted into a common parent class, for example `FormView`, but that topic will be saved for another book.

List HTML Template

Two templates are used to create the HTML table listing issues. The `issuelist.html` template is used by the `CompositeView` to render the table structure including the table column headings. The `issuelistitem.html` template defines the structure of each row in the table. Let's create a new table column named *Actions* which will contain buttons to perform activities on an item in the list.

In the `src/main/app/templates` directory, open the file named `issuelist.html`.

Issue List Template

```

1 <div class="row">
2   <div class="col-md-12">
3     <table class="table table-hover">
4       <thead>
5         <tr>
6           <th class="col-md-1">Priority</th>
7           <th class="col-md-1">Type</th>
8           <th class="col-md-1">Status</th>
9           <th class="col-md-7">Title</th>
10          <th class="col-md-2">Actions</th>
11        </tr>
12      </thead>
13      <tbody>
14      </tbody>
15    </table>
16  </div>
17 </div>

```

The changed HTML markup is highlighted in the excerpt above. Each row in the Bootstrap grid system has twelve columns. Reduce the width of the *Title* table column heading from nine Bootstrap grid columns to seven. Add the *Actions* column, allocating it the two Bootstrap grid columns that were subtracted from the *Title* column.

List Item HTML Template

The HTML markup for each row in the table body is defined in the `issuelistitem.html` template. We need to update the table body row to match the columns defined in the table header.

Open the `issuelistitem.html` source file and make the changes noted below.

Issue List Item Template

```

1 <td class="col-md-7">
2   <a href="#" class="js-view"><%= title %></a>
3 </td>
4
5 <td class="col-md-2">
6   <button type="button" class="btn btn-default btn-xs js-edit">
7     <i class="fa fa-pencil fa-fw"></i> Edit
8   </button>
9 </td>

```

Reduce the *Title* column to seven Bootstrap grid columns. Add the markup for the *Actions* table column. The column contains a button for the *Edit* action.

Empty List HTML Template

The HTML markup displayed when the issue collection is empty is defined in the `issuelis-empty.html` template. Let's update the template so that the row spans the fifth column.

Empty List Template

```

1 <td colspan="5" class="col-md-12">No issues to display.</td>

```

Change the value of the `colspan` attribute from 4 to 5 so that the single column definition is the full width of the table.

List Item View

Each row in the issue list table is managed by an instance of the `IssueListItemView` class. When a user clicks the *Edit* button in the issue list, the view instance managing that row receives the click event.

Clicking the *Edit* button should show the edit issue form for the specific issue whose *Edit* button was clicked. The issue manager controller is responsible for view transitioning, so let's trigger an event when the *Edit* button is clicked. The controller can listen for the event and execute the logic required.

After the issue model object has been updated, it triggers a change event. The `Marionette View` class provides a `modelEvents` hash, similar to the `events` hash, but dedicated to listening to events on the view's model. Let's listen for change events and render the view again to display the updated attribute values to the user.

Open the `IssueManagerViews.js` source file and update the `IssueListItemView` class.

Issue List Item View

```
1  // Define the View for a single Issue in the List
2  IssueManager.IssueListItemView = Backbone.Marionette.ItemView.extend({
3
4      tagName: 'tr',
5
6      template: 'issuelistitem',
7
8      modelEvents: {
9          'change': 'render'
10     },
11
12     triggers: {
13         'click .js-view': 'issue:view',
14         'click .js-edit': 'issue:edit'
15     }
16
17 });
```

The excerpt highlights the additions to the `IssueListItemView` class.

The `modelEvents` hash configures a listener for change events published by the view's model. When a change event is received, the `render` function is invoked on the view, regenerating the view's HTML markup using the updated model.

The `triggers` hash defines a click event listener on the HTML DOM element having CSS class `js-edit` that we added to the *Edit* button. When the button is clicked, the view triggers an event named `issue:edit`. Trigger event handler functions receive a single argument with three properties: `view`, `model`, and `collection`. These properties are the triggering view, the view's model, and the view's collection.

Controller

The request to edit an issue is initiated within the list of issues. Let's modify the issue manager controller's `list` function to handle requests to edit issues.

Open the `IssueManagerController.js` source file and update the `list` function of `IssueManagerController` as shown below.

Issue Manager Controller - List Function

```

1      list: function(collection) {
2          logger.debug("IssueManagerController.list");
3
4          var displayListView = function(issueCollection) {
5              var listView = new IssueManager.IssueListView({
6                  collection: issueCollection
7              });
8
9              // Handle 'issue:edit' events triggered by Child Views
10             listView.on('childview:issue:edit', function(args) {
11                 logger.debug("Handling 'childview:issue:edit' event");
12                 IssueTrackerApp.execute('issuemanager:edit', args.model.get('id'), arg\
13 s.model, issueCollection);
14             });
15
16             // Handle 'issue:view' events triggered by Child Views
17             listView.on('childview:issue:view', function(args) {
18                 logger.debug("Handling 'childview:issue:view' event");
19                 IssueTrackerApp.execute('issuemanager:view', args.model.get('id'), arg\
20 s.model, issueCollection);
21             });
22
23             logger.debug("Show IssueListView in IssueTrackerApp.mainRegion");
24             IssueTrackerApp.mainRegion.show(listView);
25         };
26
27         if(collection) {
28             displayListView(collection);
29         } else {
30             var fetchingIssues = IssueTrackerApp.request('issue:entities');
31             $.when(fetchingIssues).done(function(issues) {
32                 displayListView(issues);
33             });
34         }
35     },

```

The complete list function is depicted above with the new logic highlighted.

Add an event listener to listView for the issue:edit event triggered by IssueListItemView. CompositeViews have a special relationship with their child views. Events triggered by child views are republished by the CompositeView with childview: prepended to the event name. Therefore, in

the controller, we are able to handle events triggered by instances of `IssueListItemView` by listening to the `IssueListView` object.

When the `childview:issue:edit` event is received, the controller executes an application command named `issuemanager:edit`. The controller passes the issue identifier, the issue model, and the collection of issues as command parameters.

Next, create a command handler function to receive `issuemanager:edit` commands. Within the `IssueManagerController.js` source file, below the command handler for viewing issues, add the following logic.

Edit Issue Command Handler

```

1  IssueTrackerApp.commands.setHandler('issuemanager:edit',
2    function(id, model, collection) {
3      logger.debug("Handling 'issuemanager:edit' command");
4      controller.edit(id, model, collection);
5    }
6  );

```

Upon receipt of an `issuemanager:edit` command, the handler function calls a new controller function named `edit`. The command parameters are passed to the `edit` function.

Let's create the `edit` function within the `IssueManagerController` class. The logic for the `edit` function is depicted below.

IssueManagerController - Edit Function

```

1  edit: function(id, model, collection) {
2    logger.debug("IssueManagerController.edit");
3
4    var displayEditView = function(issueModel, issueCollection) {
5      var editIssueView = new IssueManager.IssueEditView({
6        model: issueModel
7      });
8
9      // Handle 'form:cancel' event
10     editIssueView.on('form:cancel', function() {
11       logger.debug("Handling 'form:cancel' event");
12       IssueTrackerApp.execute('issuemanager:view', id, issueModel, issueColl\
13     ection);
14     });
15
16     // Handle 'form:submit' event
17     editIssueView.on('form:submit', function(data) {

```

```

18         logger.debug("Handling 'form:submit' event");
19         logger.debug("form data:" + JSON.stringify(data));
20         if(issueModel.save(data,
21             {
22                 success: function() {
23                     IssueTrackerApp.execute('issuemanager:view', id, issueModel, iss\
24 ueCollection);
25                 },
26                 error: function() {
27                     alert('An unexpected problem has occurred.');


---



```

Similar to the view controller function, the edit function contains a function named `displayEditView` which performs the view instantiation and event handler logic. The `displayEditView` function is passed an issue model to be edited and an optional collection of issues if the edit operation was triggered from within the list view.

When the `form:cancel` event is triggered by the `editIssueView`, the controller transitions back to the single issue detail view.

When the `form:submit` event is triggered by the `editIssueView`, the controller manages the server interaction to update the issue entity attributes. The logic is nearly identical to the `form:submit` handler method for creating issues. Instead of creating a new `IssueTrackerApp.Entities.Issue` model instance and populating it with the serialized web form data, the existing issue entity model is used. The `save` function is called on the issue model with the updated, serialized form data. The `save` function validates the updated data and then, detecting that this model has already been created, issues an AJAX PUT request to the server to synchronize the server state with the requested changes. When the server returns successfully, the controller transitions to the single issue detail view.

After the `displayEditView` function definition, the controller logic analyzes the parameters passed to the `edit` function. If the model parameter is undefined, the model is fetched from the server using the primary key identifier and the `displayEditView` function is called with the fetched model. However, if an issue model is passed to the `edit` function, that model instance is passed to `displayEditView`.

Running the Application

Let's test the new functionality. Open a terminal window and navigate to the web services project base directory. Type `mvn spring-boot:run` and press *Enter* to start the web services on port 8080. Open a second terminal window, or tab, and navigate to the user interface project base directory. Type `grunt run` and press *Enter* to start the local web server on port 9000.

Open a browser and press F12 to open the developer tools. In the browser's address bar type `http://localhost:9000/index.html` and press *Enter*. The issue manager module displays an empty issue list. Click the add issue button in the top navigation bar and create a new issue. Click the edit button next to the issue in the list. Modify some of the issue data and click the *Update* button. If everything works correctly, an AJAX PUT request is sent to the server and the server returns a 200 status code indicating the issue has been successfully updated. The user interface displays the updated issue.

The screenshot shows a web browser window titled 'Issue Tracker' with the address bar displaying 'localhost:9000/index.html#issues'. The application header includes the 'Issue Tracker' logo, a dropdown menu for 'Issues', and a '+ Issue' button. The main content area is titled 'Edit Issue'. Below the title, there are two rows of buttons. The first row contains buttons for issue types: 'Task' (checked), 'Story', 'Enhancement', and 'Bug'. The second row contains buttons for priority levels: 'Low' (selected with a green arrow), 'Medium', and 'High'. To the right of these are buttons for issue status: 'Open', 'In Progress', and 'Done'. Below these buttons is a 'Title' field containing the text 'One Small Problem'. Underneath the title field is a 'Description' section with a text area containing the text 'I found a small problem in the application. To summarize, it's a small problem.' At the bottom of the form are two buttons: 'Update' (highlighted in blue) and 'Cancel'. The footer of the application displays the text 'A LeanStacks Solution'.

Issue Manager - Edit

In the console tab of the developer tools, log statements depict the application flow.

Console Log

```
1 10:25:13 DEBUG - IssueTrackerApp.start
2 10:25:13 DEBUG - Footer initializer
3 10:25:13 DEBUG - FooterController.show
4 10:25:13 DEBUG - Header initializer
5 10:25:13 DEBUG - HeaderController.show
6 10:25:13 DEBUG - IssueManager initializer
7 10:25:13 DEBUG - Backbone.history.start
8 10:25:13 DEBUG - Handling 'issuemanager:list' command
9 10:25:13 DEBUG - IssueTrackerApp.navigate route:issues
10 10:25:13 DEBUG - IssueManagerController.list
11 10:25:13 DEBUG - Handling 'issue:entities' request
```



```
12 10:25:13 DEBUG - IssueEntityController.getIssues
13 10:25:13 DEBUG - Show IssueListView in IssueTrackerApp.mainRegion
14 10:25:22 DEBUG - Handling 'issuemanager:add' command
15 10:25:22 DEBUG - IssueManagerController.add
16 10:25:22 DEBUG - Show IssueAddView in IssueTrackerApp.mainRegion
17 10:25:45 DEBUG - IssueAddView.onCreateClicked
18 10:25:45 DEBUG - Handling 'form:submit' trigger
19 10:25:45 DEBUG - form data:{"type":"TASK","priority":"HIGH","title":"Do This Tas\
20 k","description":"Here is what you should do..."}
21 10:25:45 DEBUG - IssueManagerController.list
22 10:25:45 DEBUG - Handling 'issue:entities' request
23 10:25:45 DEBUG - IssueEntityController.getIssues
24 10:25:45 DEBUG - Show IssueListView in IssueTrackerApp.mainRegion
25 10:25:47 DEBUG - Handling 'childview:issue:edit' trigger
26 10:25:47 DEBUG - Show IssueEditView in IssueTrackerApp.mainRegion
27 10:25:51 DEBUG - IssueEditView.onUpdateClicked
28 10:25:51 DEBUG - Handling 'form:submit' trigger
29 10:25:51 DEBUG - form data:{"type":"TASK","priority":"HIGH","status":"IN_PROGRES\
30 S","title":"Do This Task","description":"Here is what you should do..."}
31 10:25:51 DEBUG - Show IssueView in IssueTrackerApp.mainRegion
```

Lines 1 through 13 depict the Marionette application startup flow. Lines 14 through 24 are logged as the new issue is created. Line 25 is logged when the *Edit* button is clicked. The *IssueEditView* instance is shown in the main region of the application. Line 27 indicates the update issue form has been submitted. After successfully synchronizing the updated issue model with the server, the single issue detail view is shown in the main region of the application.



The Road Ahead

In the next chapter, we will enhance both our web services and user interface to delete issues.

Deleting Issues

The applications have the capability to create, edit, view, and list issues. In this chapter, we will construct a new web service to delete an issue entity from the database. We will enhance the user interface, adding the capability to delete issues from the list.

Web Services

By now, the pattern for developing RESTful web services is emerging. Each new web service requires a request handler method in the controller, a method on the service interface, and the implementation method on the service bean. There are subtle differences to the web service URI mapping annotation and to the business service logic, but the implementation of a new web service follows a standardized, repeatable approach. This simple approach translates to highly reliable and easily maintainable applications.

Service Interface

The service interface defines the contract for the service implementation class. Previously the update method was declared in the `IssueService` interface. Just below it, let's define a new method to delete Issue entities.

Issue Service - Delete

```
1  /**
2   * Delete an Issue entity from the data repository.
3   * @param id The primary key identifier of the issue to delete.
4   */
5  void delete(Long id);
```

The `delete` method receives an Issue entity primary key identifier for the entity to be deleted from the database.

Service Implementation

The service implementation class contains the functionality for methods declared on the service interface. We need to add the implementation for the `delete` method that we declared on the `IssueService` interface. Open the `IssueServiceBean` and add the following logic below the `update` method.

Issue Service Implementation - Delete

```
1  @Override
2  public void delete(Long id) {
3      logger.info("> delete");
4
5      issueRepository.delete(id);
6
7      logger.info("< delete");
8  }
```

The Spring Data framework implements all of the logic to connect to the database, generate SQL delete statements, and remove the contents of the Issue entity in the database. The repository delete method removes the persisted version of the Issue entity from the database.

Controller

The controller class implements the RESTful web service request handler methods. Let's code the request handler method and annotations that will receive requests to delete Issue entities. Open the `IssueController` class and add the following logic below the `updateIssue` request handler method.

Issue Controller - Delete

```
1  @RequestMapping(
2      value = "/issues/{id}",
3      method = RequestMethod.DELETE)
4  public ResponseEntity<Issue> deleteIssue(@PathVariable("id") Long issueId) {
5      logger.info("> deleteIssue");
6
7      try {
8          issueService.delete(issueId);
9      } catch (Exception e) {
10         logger.error("Unexpected Exception caught.", e);
11         return new ResponseEntity<Issue>(HttpStatus.INTERNAL_SERVER_ERROR);
12     }
13
14     logger.info("< deleteIssue");
15     return new ResponseEntity<Issue>(HttpStatus.NO_CONTENT);
16 }
```

The context path to delete entities is the same as that which is used to update them. Rather than just the plural entity name, `/issues`, the path also contains the primary key identifier of the entity

instance to be deleted. In the Spring Framework, context path variables are delimited by curly braces. The value of the `id` context path variable is passed in a method parameter annotated with `@PathVariable("id")`.

The HTTP method used to delete entities is `DELETE`. Therefore, to delete an issue entity whose primary key value is 1, a HTTP `DELETE` request is sent to `/issues/1`.

The request mapping annotation for deleting issues requires neither the *consumes* nor the *produces* elements. The request body does not contain any data. The method returns only a HTTP status code with an empty response body.

The web service responds with HTTP status code 204, indicating that the request completed successfully, but the service is not returning a response.

User Interface

Let's enhance the user interface to call the `deleteIssue` web service. To allow users the ability to delete issues, a *Delete* button will be added next to the *Edit* button in the issue list. When clicked, a dialog box will be displayed, prompting the user to confirm the deletion.

To realize this design, we will create a reusable Bootstrap modal dialog box using a HTML template and a `ItemView` class to manage user interaction with the dialog buttons. The list item template will be updated to create a *Delete* button for each issue in the list. The issue manager controller will be modified to orchestrate the interaction between the views and to send delete issue web service requests to the server.

Common Module

Almost every software project contains some amount of logic that is needed in multiple project components. Rather than rewriting the logic in each component, creating a maintenance nightmare, the code should be consolidated into a common location where it may be consumed by all project components.

We need to create a dialog box to confirm a user's request to delete an issue. The `ItemView` could be created within the issue manager module since it is needed to delete issues. However, it is highly likely that some future requirement will necessitate the use of another dialog box. To promote future reuse of the dialog box, let's create a new Marionette module, the *Common* module, to house shared, reusable components.

Dialog HTML Template

A [Bootstrap modal dialog](#)⁵⁹ is comprised of HTML markup and JavaScript behavior. The HTML markup defines the header, body, and footer sections of the dialog.

⁵⁹<http://getbootstrap.com/javascript/#modals>

In the `src/main/app/templates` directory, create a new file named `dialog.html` and place the following Bootstrap modal dialog markup within it.

Dialog Template

```
1 <div class="modal-dialog">
2
3   <div class="modal-content">
4
5     <div class="modal-header">
6       <h4 class="modal-title"><%= title %></h4>
7     </div>
8
9     <div class="modal-body">
10      <p><%= body %></p>
11    </div>
12
13    <div class="modal-footer">
14      <button type="button"
15        class="btn btn-default js-secondary">
16        <%= secondary %>
17      </button>
18      <button type="button"
19        class="btn btn-primary js-primary">
20        <%= primary %>
21      </button>
22    </div>
23
24  </div>
25
26 </div>
```

Bootstrap modal dialog boxes are wrapped in three `div` tags. The outermost `div` tag will be generated by the dialog's `ItemView` class. This structure is used by the Bootstrap framework to manage behavior and apply visual styles to the dialog box as state changes occur.

The content of the modal dialog box is divided into a header, body, and footer. The header and body contain underscore template markup so that the title and body text is dynamically generated using data provided by the `Dialog ItemView`.

The footer contains two buttons. To promote reuse, we refer to the buttons as *primary* and *secondary* rather than prescribing the meaning of the button within the template. The CSS classes `js-primary` and `js-secondary` are used by the dialog's `ItemView` class to handle user interaction with the buttons. Like the title and body text, each button label is dynamically rendered using data supplied to the template.

Dialog View

Let's create the view class to render the Bootstrap modal dialog box and manage user interaction with it. As previously discussed, the view will be created in a new module named *Common*. In the `src/main/app/js/modules` directory, create a new subdirectory named `common`. Within the `common` module directory, create a source file named `CommonViews.js` and include the logic below.

Common Module - Dialog View

```
1 IssueTrackerApp.module('Common', function(Common, IssueTrackerApp, Backbone, Mar\
2 ionette, $, _) {
3
4   // Define the View for a Modal Dialog Box
5   Common.DialogView = Backbone.Marionette.ItemView.extend({
6
7     className: 'modal fade',
8
9     template: 'dialog',
10
11    triggers: {
12      'click .js-primary': 'primary',
13      'click .js-secondary': 'secondary'
14    },
15
16    serializeData: function() {
17      return this.options;
18    },
19
20    onShow: function() {
21      this.$el.modal('show');
22    },
23
24    onHide: function() {
25      this.$el.modal('hide');
26    }
27  });
28 });
29
30 });
```

The first line declares the `Common` module on our Marionette application, `IssueTrackerApp`, using the module definition callback function.

Inside the module definition function, the `DialogView` class is declared, extending `ItemView`. The default HTML DOM element of the view is `div`. The `className` attribute ensures that the base HTML DOM element for this view is `<div class="modal fade"></div>`, which is the outermost wrapper of a Bootstrap modal dialog.

The `triggers` hash maps click events on the two dialog buttons to events triggered by the `DialogView`. When a user clicks the primary dialog button, the view publishes an event named `primary`. Similarly, clicking the secondary button triggers an event named `secondary`. The component which creates instances of `DialogView` may listen for these events and execute behavior based upon the user selection.

The `serializeData` function supplies the data used to render the template into HTML. The default implementation of `serializeData` returns the contents of the view's `model` attribute converted to JSON. Since `DialogView` does not have a `model` we override the `serializeData` function to return the `options` hash instead. The values for the dialog title, body, and button labels will be supplied via the `options` hash provided when an instance of `DialogView` is created.

The `onShow` function is called when a view is *shown* in the browser. For example, when a view instance is passed to the `show` function of a region, the `onShow` function is triggered on the view. By default, Bootstrap modal dialog boxes are hidden, even when the HTML markup is attached to the page. In the `DialogView` `onShow` function, the `Bootstrap.modal` function is called to make the modal dialog box visible to the user.

The `onHide` function also invokes the Bootstrap `modal` function, but makes the modal dialog box invisible to the user.

Issue Manager Module

The issue manager module encapsulates all activities related to issues. To create the capability to delete issues, we need to modify the list item template to add a *Delete* button and the list item view to handle click events on that button. We will also enhance the controller to manage the dialog view and process requests to delete the issue entity.

List Item Template

The list item template renders the HTML for each row in the list of issues. Let's add the *Delete* button to the actions column.

In `src/main/app/templates`, open the `issuelistitem.html` source file. Add the *Delete* button after the *Edit* button.

Issue List Item Template

```

1 <td class="col-md-2">
2   <button type="button" class="btn btn-default btn-xs js-edit">
3     <i class="fa fa-pencil fa-fw"></i> Edit
4   </button>
5   <button type="button" class="btn btn-danger btn-xs js-delete">
6     <i class="fa fa-trash fa-fw"></i> Delete
7   </button>
8 </td>

```

The snippet above shows the actions column HTML markup. The button tag for *Delete* is inserted after the *Edit* button.

The `btn-danger` CSS class produces a red button, differentiating it from the *Edit* button which uses the default button color.

The `js-delete` CSS class will be used by the issue list item view to handle user interaction with the *Delete* button.

List Item View

Let's update the issue list item view class to handle click events on the *Delete* button. Open the `IssueManagerViews.js` source file and locate the `IssueListItemView`. Update the `triggers` hash as illustrated below.

Issue List Item View

```

1   triggers: {
2     'click .js-view': 'issue:view',
3     'click .js-edit': 'issue:edit',
4     'click .js-delete': 'issue:delete'
5   }

```

In the `triggers` hash, add a DOM event listener for click events on the element with CSS class `js-delete`, which is our *Delete* button. When a click is detected, the view triggers an event named `issue:delete`.

Controller

The list item view triggers `issue:delete` events when the *Delete* button is clicked. The controller will be enhanced to orchestrate view transitioning and entity model changes when the event is published.

Let's modify the `list` function, adding an event handler function for the `childview:issue:delete` event. Open the `IssueManagerController.js` source file and insert the logic shown below immediately after the listener for the `issue:edit` event.

Issue Manager Controller

```

1      // Handle 'issue:delete' events triggered by Child Views
2      listView.on('childview:issue:delete', function(args) {
3          logger.debug("Handling 'childview:issue:delete' trigger");
4          var dialogView = new IssueTrackerApp.Common.DialogView({
5              title: "Delete Issue?",
6              body: "Click confirm to permanently delete this issue.",
7              primary: "Confirm",
8              secondary: "Cancel"
9          });
10
11         dialogView.on("primary", function() {
12             logger.debug("Handling 'primary' dialog event");
13             args.model.destroy();
14             dialogView.triggerMethod('hide');
15         });
16
17         dialogView.on("secondary", function() {
18             logger.debug("Handling 'secondary' dialog event");
19             dialogView.triggerMethod('hide');
20         });
21
22         IssueTrackerApp.dialogRegion.show(dialogView);
23     });

```

Add a listener for `childview:issue:delete` events triggered by list item view instances. The event handler function creates an instance of the `DialogView` class passing an options hash object to the constructor. The hash contains the values for variables rendered on the modal dialog box.

Next, create event listener functions for the primary and secondary buttons on the `DialogView` instance. By implementing the logic for the dialog buttons in the controller instead of the view, the `DialogView` class does not contain logic that deletes issues. It may be reused in the future, associating any desired logic to the primary and secondary buttons.

The primary event handler function invokes the `destroy` function on the entity model whose *Delete* button was clicked in the list. The `destroy` function is implemented by the `Backbone.Model` class. The model sends the HTTP DELETE request to the server and publishes a `destroy` event. The issue collection listens for `destroy` events on its models, removing the model from the collection when it is received. In turn, the issue list view detects the model has been removed from the collection and destroys the list item view corresponding to the deleted model. After destroying the model, the primary event handler function triggers the `hide` event on the `DialogView`, invoking the view's `onHide` method.

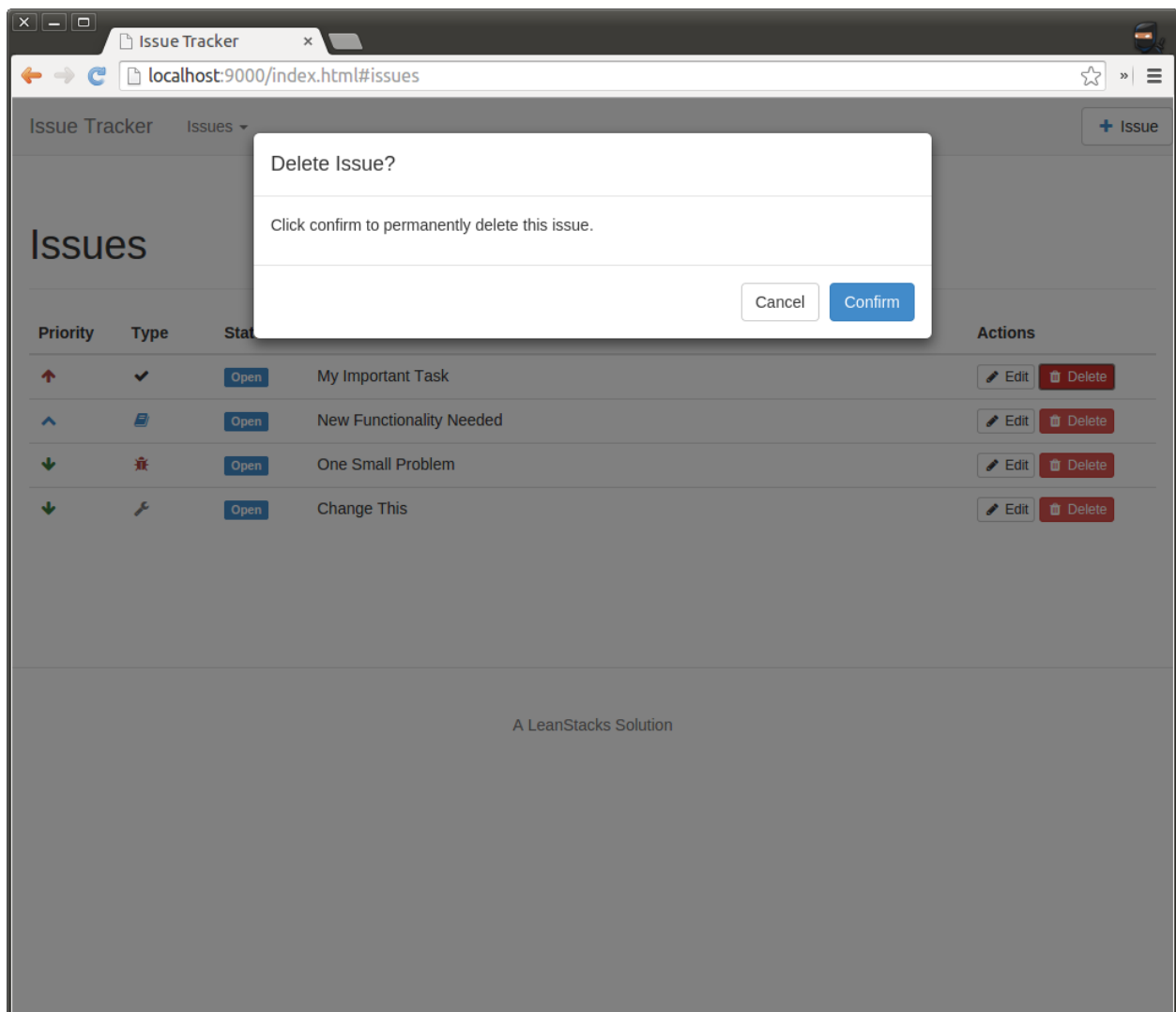
The secondary event is triggered when a user clicks the *Cancel* button on the modal dialog box. The event handler function simply hides the dialog view.

After configuring the event listeners, the dialog view is shown in the `dialogRegion` of the application. The show event triggers the execution of the `onShow` method in `DialogView` resulting in the display of the Bootstrap modal dialog box.

Running the Application

Let's test the new functionality. Open a terminal window and navigate to the web services project base directory. Type `mvn spring-boot:run` and press *Enter* to start the web services on port 8080. Open a second terminal window, or tab, and navigate to the user interface project base directory. Type `grunt run` and press *Enter* to start the local web server on port 9000.

Open a browser and press F12 to open the developer tools. In the browser's address bar type `http://localhost:9000/index.html` and press *Enter*. The issue manager module displays an empty issue list. Click the add issue button in the top navigation bar and create a new issue. Click the delete button next to the issue in the list. The dialog box is shown. Click the *Cancel* button to dismiss the dialog box. Click the *Confirm* button to delete the issue. If everything works correctly, an AJAX DELETE request is sent to the server and the server returns a 204 status code indicating the issue has been successfully deleted. The user interface returns to the issue list.



Issue Manager - Delete

In the console tab of the developer tools, you will see log statements depicting the application flow. The logging statements describing application startup and issue creation have been omitted for brevity.

Console Log

- 1 08:40:06 DEBUG - Handling 'childview:issue:delete' trigger
- 2 08:40:08 DEBUG - Handling 'primary' dialog event
- 3 08:40:13 DEBUG - Handling 'childview:issue:delete' trigger
- 4 08:40:15 DEBUG - Handling 'secondary' dialog event

Line 1 is shown when a user clicks the *Delete* button in the issue list. Line 2 indicates a user has clicked the primary modal dialog button. In this case, that is the *Confirm* button. Lines 3 and 4 depict showing the dialog box and clicking the *Cancel* button.



The Road Ahead

In the next several chapters, we will discuss techniques to make the web services and user interface applications ready for production operation.

Part V. Production Features

Development Operations

The technologies used in this book are not only *lean* application development frameworks. Both Spring Boot and Backbone.Marionette applications are *lean* development operations, or DevOps, and production operations, or ProdOps, technologies as well.

Development operations is the information technology discipline responsible for building, packaging, and deploying applications. These activities are sometimes referred to as continuous integration and continuous deployment. Many software applications require a multi-step build and assembly process, sometimes with manual steps interwoven with the automation. This complex process is brittle, often resulting in incorrectly assembled artifacts. The key to reliable and repeatable application building, packaging, deployment, and execution is simplicity. Using Maven and Grunt, the web services and user interface project artifacts are prepared for server deployment using a single command which may be easily automated using a continuous integration tool like [Jenkins](http://jenkins-ci.org/)⁶⁰.

Production operations refers to the monitoring and management of information technology assets in the production, or user-facing, environment. Spring Boot's actuator module provides out-of-the-box metrics and status capabilities accessible via multiple protocols. Actuator endpoints may be secured and customized to return tailored information. The Spring Boot actuator module will be discussed in detail in the [Production Operations](#) chapter.

Web Services

The web services project uses Apache Maven for project compilation and assembly workflow automation. The project POM file includes the `spring-boot-maven-plugin`. The plugin defines a Maven goal named `repackage` whose job is to create an executable jar file.

Spring Boot Maven Plugin pom.xml

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

⁶⁰<http://jenkins-ci.org/>



Binding to repackage

Spring Boot projects that use the `spring-boot-starter-parent` POM automatically bind the Maven `package` goal to the Spring Boot plugin `repackage` goal. If you do not use the `spring-boot-starter-parent`, you will need to create this binding.

Packaging an Executable Jar

Executable jar files contain the compiled classes from the project and all of the jar dependencies. If the `spring-boot-starter-web` dependency is declared in the POM, an Apache Tomcat web server is included in the executable jar file as well.

To create an executable jar file, open a terminal window and navigate to the base project directory where the `pom.xml` file is located. Type `mvn clean package` and press *Enter*.

Console Log

```
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building issuetracker-ws 1.0.0-SNAPSHOT
[INFO] -----
[INFO] .....
[INFO] ..... (omitted log output)
[INFO] .....
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ issuetracker-ws ---
[INFO] Building jar: target/issuetracker-ws-1.0.0-SNAPSHOT.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:1.2.4.RELEASE:repackage @ issuetracker-ws -
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
```

The console output from the Maven command is depicted above. The output is verbose and some of the lines have been omitted for brevity. Maven logs the name of the jar file as ... `target/issuetracker-ws-1.0.0-SNAPSHOT.jar`. If you navigate to the `/target` directory, the executable jar file is listed. The jar file will be approximately 25 Mb in size due to the included dependencies and embedded Apache Tomcat server.

To stop the application, including the publication of standard Spring shutdown events, simply press `ctrl-c`.

Sample Service Script

Hosting applications in a server environment requires a slightly different approach than running on a developer machine. The application should not be started from the command line requiring the terminal window left open or an active user session. Not only is this devoid of automation, but is susceptible to failure.

Server applications are typically installed as services in the operating system. Those services are automatically started and stopped as the server transitions through boot and shutdown sequences. A sample Ubuntu Linux service descriptor is illustrated below. This script is meant to be a simple, but functional example which may be used as a starting point to host the Issue Tracker web services project on an Ubuntu server.

Sample Ubuntu Service Descriptor

```

1  #!/bin/bash
2  ### BEGIN INIT INFO
3  # Provides:          issuetracker
4  # Required-Start:    $remote_fs $syslog
5  # Required-Stop:     $remote_fs $syslog
6  # Default-Start:     2 3 4 5
7  # Default-Stop:      0 1 6
8  # Short-Description: Spring Boot IssueTracker Application Init Script
9  # Description:       This file should be used to manage the Spring Boot
10 #                    IssueTracker application and is placed in /etc/init.d.
11 ### END INIT INFO
12
13 case $1 in
14     start)
15         $( cd /var/lib/issuetracker; nohup java -jar issuetracker-ws-1.0.0-SNAPSHOT.\
16 jar --spring.profiles.active=qa > /var/log/issuetracker/application.log 2>&1 & )
17         sleep 5s
18         PID=$( pgrep -f issuetracker-ws-1.0.0-SNAPSHOT.jar )
19         echo "${PID}" > /var/run/issuetracker.pid
20         ;;
21     stop)
22         kill `cat /var/run/issuetracker.pid`
23         ;;
24     *)
25         echo "usage: issuetracker {start|stop} {nn}"
26         ;;

```

```
27  esac
28  exit 0
```

Ubuntu service descriptors are located in the `/etc/init.d` directory. The scripts are typically written in bash or sh syntax.

The file begins with an `INIT` comment block. When the service is installed or removed from the server, the `INIT` block is parsed to obtain the service script metadata.

The `Provides` statement defines the name of the service.

The `Required-Start` and `Required-Stop` statements define the system services which must be running to start or stop the service.

The `Default-Start` and `Default-Stop` statements indicate when the service should be started or stopped. Linux systems have seven system states numbered 0 through 6. System states 2 through 5 indicate the server is booting or running. System states 0, 1, and 6 indicate the server is shutting down or stopped.

The script logic to start and stop the service is authored after the `INIT` block. Operations staff issue terminal commands such as: `service issuetracker start` or `service issuetracker stop` to interact with system services. The script uses a case statement to evaluate the action requested.

To start the service, the script navigates to the directory where the executable jar is located. The `nohup` command is used to place the java process in the background. A command line argument defining the active Spring profiles, `--spring.profiles.active=qa`, is passed to the application to facilitate environment-specific configuration. Using Spring profiles for externalized application configuration will be discussed in a later chapter. Finally, the standard and error output, which includes the console logging, is diverted to a file named `/var/log/issuetracker/application.log`. To allow time for the server to initiate the thread in which the Java process will execute, the script pauses for 5 seconds. The PID, or process identifier, for the Java process is queried from the server and written to a file named `/var/run/issuetracker.pid`.

To stop the service, the script uses the `kill` command with the PID written to `/var/run/issuetracker.pid` when the service started. The `kill` command works similarly to `ctrl-c` on a developer machine. The Spring Boot application gracefully shuts down.

To install an Ubuntu service, place the script in the `/etc/init.d` directory. The script's permissions should be 755. The script should be owned by root and in the root group. Use the following command to install the `issuetracker` service on an Ubuntu server:

```
$ update-rc.d issuetracker defaults
```

To uninstall the `issuetracker` service on an Ubuntu server, use this command:

```
$ update-rc.d issuetracker remove
```

User Interface

The user interface project employs Grunt for project compilation and assembly workflow automation. The Gruntfile.js file defines various tasks to automate the development process. Grunt tasks define the workflow for common activities.

Gruntfile.js Tasks

```
// Define Tasks
grunt.registerTask('default', [ 'jshint', 'clean', 'copy', 'jst', 'uglify' ]);
grunt.registerTask('dist', [ 'default', 'compress' ]);
grunt.registerTask('run', [ 'default', 'connect:server', 'watch' ]);
```

The dist task is used to prepare the project for distribution to a server environment.

Packaging a Distribution

The application consists of several files including HTML, CSS, and application and third-party JavaScript. Packaging the application into a single, portable distribution file requires a single command. Prepare the application files and create a single, compressed tar.gz distribution file using the Grunt dist task.



tar vs. tar.gz vs. zip

A tar file is predominantly used in Unix based operating systems to store the contents of a file structure into a single file. When gzip compression is applied to a tar file, the result is a .tar.gz file, commonly called “tar gunzip”.

A zip file is essentially the same as a tar gunzip; however, a zip file does not preserve file permissions in the archive and a tar gunzip does.

To prepare the user interface for distribution, open a terminal and navigate to the base project directory where the Gruntfile.js file is located. Type `grunt dist` and press *Enter*.

Console Log

```
$ grunt dist
```

```
Running "jshint:gruntfile" (jshint) task
```

```
>> 1 file lint free.
```

```
Running "jshint:main" (jshint) task
```

```
>> 9 files lint free.
```

```
Running "clean:src" (clean) task
```

```
>> 2 paths cleaned.
```

```
Running "copy:main" (copy) task
```

```
Created 10 directories, copied 33 files
```

```
Running "jst:compile" (jst) task
```

```
File dist/assets/app/js/app-templates-1.0.0.js created.
```

```
Running "uglify:main" (uglify) task
```

```
>> 1 sourcemap created.
```

```
>> 1 file created.
```

```
Running "compress:main" (compress) task
```

```
Created dist/issuetracker.tar.gz (761818 bytes)
```

```
Done, without errors.
```

The console output from the Grunt command is depicted above. Grunt uses JS Lint to review the JavaScript source for mistakes; cleans the `/dist` directory; copies source files to `/dist`; compiles the HTML templates into JavaScript; minifies and uglifies the JavaScript; and, finally, creates the distribution file. The `/dist` directory contains the prepared application files.

Listing of `/dist`

```
dist
```

```
├─ assets
```

```
│   └─ app
```

```
│       └─ css
```

```
│           └─ app.css
```

```
│       └─ js
```

```
│           └─ app-1.0.0.map
```

```
│           └─ app-1.0.0.min.js
```

```

|   |   └─ app-templates-1.0.0.js
|   └─ lib
|       ├── backbone-1.2.1.min.js
|       ├── backbone.marionette-2.4.2.min.js
|       ├── backbone.marionette.map
|       ├── backbone-min.map
|       ├── backbone.syphon-0.5.0.min.js
|       ├── backbone.syphon.min.map
|       ├── bootstrap-3.3.5
|       |   ├── css
|       |   |   ├── bootstrap.css
|       |   |   ├── bootstrap.css.map
|       |   |   ├── bootstrap.min.css
|       |   |   ├── bootstrap-theme.css
|       |   |   ├── bootstrap-theme.css.map
|       |   |   └─ bootstrap-theme.min.css
|       |   ├── fonts
|       |   |   ├── glyphsicons-halflings-regular.eot
|       |   |   ├── glyphsicons-halflings-regular.svg
|       |   |   ├── glyphsicons-halflings-regular.ttf
|       |   |   ├── glyphsicons-halflings-regular.woff
|       |   |   └─ glyphsicons-halflings-regular.woff2
|       |   └─ js
|       |       ├── bootstrap.js
|       |       └─ bootstrap.min.js
|       ├── fontawesome-4.3.0
|       |   ├── css
|       |   |   ├── font-awesome.css
|       |   |   └─ font-awesome.min.css
|       |   └─ fonts
|       |       ├── FontAwesome.otf
|       |       ├── fontawesome-webfont.eot
|       |       ├── fontawesome-webfont.svg
|       |       ├── fontawesome-webfont.ttf
|       |       └─ fontawesome-webfont.woff
|       ├── jquery-2.1.4.min.js
|       ├── jquery-2.1.4.min.map
|       ├── json2.js
|       ├── log4javascript-1.4.13.js
|       ├── underscore-1.8.3.min.js
|       └─ underscore-min.map
└─ index.html

```

```
└─ issueracker.tar.gz
```

To ensure the archive contains all of the required files, type `tar -tvf issueracker.tar.gz` and press *Enter* to view a listing of the archive file contents.

Listing of issueracker.tar.gz

```
$ tar -tvf issueracker.tar.gz
drwxrwxr-x user/user      0 2014-12-03 09:25 assets/
drwxrwxr-x user/user      0 2014-12-03 09:25 assets/lib/
-rw-rw-r-- user/user    15626 2014-10-20 07:35 assets/lib/underscore-1.8.3.min.\
js
-rw-rw-r-- user/user    128349 2014-09-28 13:33 assets/lib/log4javascript-1.4.13\
.js
-rw-rw-r-- user/user    141680 2014-10-20 07:37 assets/lib/jquery-2.1.4.min.map
drwxrwxr-x user/user      0 2014-12-03 09:25 assets/lib/bootstrap-3.3.5/
drwxrwxr-x user/user      0 2014-12-03 09:25 assets/lib/bootstrap-3.3.5/fonts/
-rw-rw-r-- user/user     62927 2014-06-26 12:14 assets/lib/bootstrap-3.3.5/fonts\
/glyphicons-halflings-regular.svg
-rw-rw-r-- user/user     23320 2014-06-26 12:14 assets/lib/bootstrap-3.3.5/fonts\
/glyphicons-halflings-regular.woff
-rw-rw-r-- user/user     23320 2014-06-26 12:14 assets/lib/bootstrap-3.3.5/fonts\
/glyphicons-halflings-regular.woff2
-rw-rw-r-- user/user     20335 2014-06-26 12:14 assets/lib/bootstrap-3.3.5/fonts\
/glyphicons-halflings-regular.eot
-rw-rw-r-- user/user     41280 2014-06-26 12:14 assets/lib/bootstrap-3.3.5/fonts\
/glyphicons-halflings-regular.ttf
drwxrwxr-x user/user      0 2014-12-03 09:25 assets/lib/bootstrap-3.3.5/js/
-rw-rw-r-- user/user     60681 2014-06-26 12:14 assets/lib/bootstrap-3.3.5/js/bo\
otstrap.js
-rw-rw-r-- user/user     31819 2014-06-26 12:14 assets/lib/bootstrap-3.3.5/js/bo\
otstrap.min.js
drwxrwxr-x user/user      0 2014-12-03 09:25 assets/lib/bootstrap-3.3.5/css/
-rw-rw-r-- user/user     23071 2014-06-26 12:14 assets/lib/bootstrap-3.3.5/css/b\
ootstrap-theme.css.map
-rw-rw-r-- user/user     21368 2014-06-26 12:14 assets/lib/bootstrap-3.3.5/css/b\
ootstrap-theme.css
-rw-rw-r-- user/user    109518 2014-06-26 12:14 assets/lib/bootstrap-3.3.5/css/b\
ootstrap.min.css
-rw-rw-r-- user/user     18860 2014-06-26 12:14 assets/lib/bootstrap-3.3.5/css/b\
ootstrap-theme.min.css
-rw-rw-r-- user/user    220790 2014-06-26 12:14 assets/lib/bootstrap-3.3.5/css/b\
ootstrap.css.map
```

-rw-rw-r-- user/user	132546	2014-06-26 12:14	assets/lib/bootstrap-3.3.5/css/b\
ootstrap.css			
-rw-rw-r-- user/user	19999	2014-10-20 07:28	assets/lib/backbone-1.2.1.min.js
-rw-rw-r-- user/user	5532	2014-07-20 13:18	assets/lib/backbone.syphon.min.m\
ap			
-rw-rw-r-- user/user	25382	2014-10-20 07:35	assets/lib/underscore-min.map
drwxrwxr-x user/user	0	2014-12-03 09:25	assets/lib/fontawesome-4.3.0/
drwxrwxr-x user/user	0	2014-12-03 09:25	assets/lib/fontawesome-4.3.0/fon\
ts/			
-rw-rw-r-- user/user	85908	2014-08-26 12:46	assets/lib/fontawesome-4.3.0/fon\
ts/FontAwesome.otf			
-rw-rw-r-- user/user	287007	2014-08-26 12:46	assets/lib/fontawesome-4.3.0/fon\
ts/fontawesome-webfont.svg			
-rw-rw-r-- user/user	56006	2014-08-26 12:46	assets/lib/fontawesome-4.3.0/fon\
ts/fontawesome-webfont.eot			
-rw-rw-r-- user/user	112160	2014-08-26 12:46	assets/lib/fontawesome-4.3.0/fon\
ts/fontawesome-webfont.ttf			
-rw-rw-r-- user/user	65452	2014-08-26 12:46	assets/lib/fontawesome-4.3.0/fon\
ts/fontawesome-webfont.woff			
drwxrwxr-x user/user	0	2014-12-03 09:25	assets/lib/fontawesome-4.3.0/css/
-rw-rw-r-- user/user	21984	2014-08-26 12:46	assets/lib/fontawesome-4.3.0/css\
/font-awesome.min.css			
-rw-rw-r-- user/user	26651	2014-08-26 12:46	assets/lib/fontawesome-4.3.0/css\
/font-awesome.css			
-rw-rw-r-- user/user	45496	2014-09-15 21:30	assets/lib/backbone.marionette.m\
ap			
-rw-rw-r-- user/user	17569	2014-09-09 18:39	assets/lib/json2.js
-rw-rw-r-- user/user	3569	2014-07-20 13:18	assets/lib/backbone.syphon-0.5.0\
.min.js			
-rw-rw-r-- user/user	95786	2014-10-20 07:36	assets/lib/jquery-2.1.4.min.js
-rw-rw-r-- user/user	38881	2014-09-15 21:30	assets/lib/backbone.marionette-2\
.4.2.min.js			
-rw-rw-r-- user/user	28133	2014-10-20 07:28	assets/lib/backbone-min.map
drwxrwxr-x user/user	0	2014-12-03 09:25	assets/app/
drwxrwxr-x user/user	0	2014-12-03 09:25	assets/app/js/
-rw-rw-r-- user/user	10867	2014-12-03 09:25	assets/app/js/app-0.1.0.map
-rw-rw-r-- user/user	22326	2014-12-03 09:25	assets/app/js/app-templates-0.1.\
0.js			
-rw-rw-r-- user/user	10312	2014-12-03 09:25	assets/app/js/app-0.1.0.min.js
drwxrwxr-x user/user	0	2014-12-03 09:25	assets/app/css/
-rw-rw-r-- user/user	528	2014-11-24 06:02	assets/app/css/app.css
-rw-rw-r-- user/user	1248	2014-11-20 06:25	index.html

The user interface project is now packaged in a single artifact ready for deployment to a web server.

Web Server Options

There are many open source web servers that can host Backbone.Marionette applications. [Apache](http://httpd.apache.org/)⁶¹, [Nginx](http://nginx.org/en/)⁶², and [node.js](http://nodejs.org/)⁶³ are a few of the popular tools available.

When selecting a web server to host an application, it is best to follow a traditional product selection process. Make a list of feature requirements. Prioritize the features and denote those which are “must have” versus “nice to have”. Then score the candidates using the feature matrix to determine the best solution for your needs.

Deployment Strategies

The complete Issue Tracker application consists of two deployable artifacts. The user interface project packaged as `issuetracker.tar.gz`. And the web services project packaged as `issuetracker-ws-1.0.0-SNAPSHOT.jar`. There are a few options to consider when deploying the application components to host servers. Each has advantages and disadvantages.

To illustrate the differences in the approaches, let’s assume the Issue Tracker application Service Level Agreement, or SLA, specifies the application must handle 500 concurrent requests continuously with a peak load of 1,000 concurrent requests. Furthermore, the web services must respond in less than 1 second 99.9% of the time.

Single Server Model

In the single application server model each application artifact is deployed the same server, or server archetype. The specifications of the host server are estimated from the sum requirements of the components hosted on the server. In the simplest environment, the user interface and web services are hosted on a single server. For environments requiring greater capacity and fault tolerance, the components are hosted on a load balanced set of server nodes. Each server node hosts both the user interface and web services.

This strategy requires that the infrastructure be scaled to meet the needs of the most resource intensive component. Suppose the user interface web server can handle four times the number of concurrent requests as the web services. Since the application components are deployed to the same application server, there will be four times the number of user interface web server instances than what is needed to meet capacity requirements.

This approach may restrict future operational flexibility. Suppose a new web service client is constructed in the future. The servers hosting the web services may need to be scaled to allow

⁶¹<http://httpd.apache.org/>

⁶²<http://nginx.org/en/>

⁶³<http://nodejs.org/>

for the increased load due to the new client. Since the user interface is deployed to the same server, the number of instances are scaled as well. This may seem trivial; however, it also means that future user interface deployments will need to be applied to a greater number of host servers.

This model typically has somewhat less operational cost. The strategy employs fewer load balancers and servers.

Let's apply the hypothetical service level agreement numbers to the single application server model to determine a rough cost. One user interface web server can handle a peak load of 1,000 concurrent connections without performance degradation. One web services web server can handle a load of 250 concurrent connections, but performance degrades beyond acceptable limits with more connections. We need four web services servers to handle a peak load of 1,000 concurrent connections and add one additional server for fault tolerance. Therefore, we require a total of five application servers.

Sample Operational Costs

Type	Quantity	Unit Cost	Total Cost
Load Balancer	1	25.00	25.00
Application Server	5	200.00	1000.00
Database Server	2	1000.00	2000.00
			\$3,025.00

Assumptions:

- The application server hosts a Java process requiring moderate CPU and memory it requires little I/O.
- The application server also hosts a web server for the user interface, but it does not require an increase in application server size.
- The database server has high CPU and memory requirements as well as a high I/O storage medium such as SSD.
- The database engine is MySQL implemented on two servers with a master/slave relationship for recoverability.

Let's compare this strategy to one where each application component is hosted on a dedicated server.

Dedicated Server Model

In the dedicated application server model each application artifact is deployed to its own distinct server, or server archetype. The choice of host server may be more finely tuned to the individual application component it hosts rather than the sum requirements of multiple components. In the simplest environment, the user interface is hosted on one server and the web services on another. For an environment requiring greater capacity and fault tolerance, the user interface is hosted on a load balanced set of server nodes and the web services are hosted on a second.

This strategy allows each application component to be scaled independently of the other. Suppose the user interface web server can handle four times the number of concurrent requests as the web services. Since the application components are deployed to different load balanced sets of servers, we can scale the web service host servers to increase capacity without growing the pool of user interface host servers.

This approach also provides greater operational flexibility. As the complexity of the application grows, it is valuable to be able to manage each application component independently. Suppose a new web service client is constructed in the future. The servers hosting the web services may be scaled to allow for the increased load due to the new client while keeping the number of user interface web servers unchanged.

However, this model typically has somewhat greater operational cost. The strategy employs twice the number of load balancers as the single layer approach. There are also a greater number of application servers. It is important to remember that every server provisioned must also be maintained by a systems administrator. These *internal* costs are often overlooked.

Let's apply the hypothetical service level agreement numbers to the dedicated server model to determine a rough cost. One user interface server can handle a peak load of 1,000 concurrent connections without performance degradation. For fault tolerance, we will use two user interface servers. One web services server can handle a load of 250 concurrent connections, but performance degrades beyond acceptable limits with more connections. We need four web services servers to handle a peak load of 1,000 concurrent connections and add one additional server for fault tolerance.

Sample Operational Costs

Type	Quantity	Unit Cost	Total Cost
Load Balancer	2	25.00	50.00
User Interface Server	2	100.00	200.00
Web Services Server	5	200.00	1000.00
Database Server	2	1000.00	2000.00
			\$3,250.00

Assumptions:

- The user interface host server requires little I/O, memory, and CPU.
- The web services server hosts a Java process requiring more CPU and memory than the user interface server; however, it too requires little I/O.
- The database server has high CPU and memory requirements as well as a high I/O storage medium such as SSD.
- The database engine is MySQL implemented on two servers with a master/slave relationship for recoverability.



Replication: MongoDB vs MySQL

Production MongoDB installations should be deployed as a replica set to keep the data synchronized across multiple server nodes. The minimum recommended number of servers in a MongoDB replica set is three. When calculating operational costs, ensure database replication costs are captured.

This hypothetical calculation illustrates that the dedicated server strategy has slightly higher infrastructure cost. There is one additional load balancer and two additional servers that require operational oversight and maintenance.

In summary, this strategy has a bit more cost and complexity for a small company building its first application. However, the long term benefits of deploying each application component to a dedicated server archetype outweighs the short-term costs.

Configuration

Transitioning a new software project from the local-only development environment to server deployment is a not a trivial exercise. In the fast paced environment of prototyping and building the alpha release, application configuration is often hard coded. As good software engineers, we add TODO comments throughout our code as a reminder to externalize configuration values later. Too often, later comes sooner than we would like.

Fortunately, the Spring Boot framework implements all of the boilerplate code to externalize configuration into property files. The property values may be injected directly into managed beans using the `@Value` annotation or accessed via the [Environment](#)⁶⁴ abstraction.

Properties whose values change depending upon the environment to which the application is deployed are known as *environment-specific* properties. Using Spring profiles, environment-specific property values replace the defaults, while retaining the property values excluded from the profile.

As an alternative or supplement to profiles, Spring Boot has a specific property source loading order. Property values may be overridden simply by supplying the desired configuration in a location which overrides the default configuration values.

Property Files

Property files have long been the preferred mechanism for externalizing Java application configuration. The Spring Boot property source loader searches for a configuration file named `application.properties` by default.



Configuration File Name

The file name may be changed by setting an environment variable or passing a command line argument named `spring.config.name`.

The property source loader searches for `application.properties` within the jar file located at the classpath root or the classpath `/config` package. Let's create a configuration file that will be packaged in the classpath `/config` package. Open the web services project and navigate to the `src/main` directory. Create a new sub-directory of `main` named `resources` and then within `resources` create another sub-directory named `config`. Within the `src/main/resources/config` directory, create the `application.properties` file.

⁶⁴<http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/core/env/Environment.html>

Maven automatically packages the contents of the `src/main/resources` project directory at the root of a jar or war file classpath. Since the `application.properties` file is nested within the `/config` sub-directory, `/config/application.properties` will be located at the classpath root. The Spring property source loader will automatically discover the file and make the configuration values available to the application.

To ensure the configuration is loaded properly, define the following property in the configuration file.

`application.properties`

```
1 ##
2 # Embedded Server
3 ##
4 server.port=9080
```

Spring Boot has many predefined properties which are initialized to sensible default values. The `server.port` property may be used to set the port on which the embedded server listens for HTTP requests. The default value is 8080. Start the web services project by running `mvn spring-boot:run` at a command prompt. Near the end of the console log, Apache Tomcat prints the HTTP port.

Console Log

```
1 2014-12-02 10:16:34.961 INFO 13819 --- [          main] s.b.c.e.t.TomcatEmbedd\
2 edServletContainer : Tomcat started on port(s): 9080/http
```

The server has started on port 9080 proving the test successful. Maven correctly packaged the `application.properties` file and the application automatically loaded and applied the property values.

To utilize properties within an application component, use the `@Value` annotation or the `Environment` abstraction. Suppose we wish to set a default value for an issue title if one is not supplied by the user. The `application.properties` file contains:

`application.properties` with Issue title

```
1 ##
2 # Issue Service
3 ##
4 issue.title.default=I Was Forgotten
```

The `issue.title.default` property value may be injected into any Spring bean using the `@Value` annotation like this:

Property Value Injection

```
1 @Service
2 public class IssueServiceBean implements IssueService {
3
4     @Value("${issue.title.default}")
5     private String defaultTitle;
6
7     // ...
8
9 }
```

Property values may also be obtained using the Spring Environment abstraction like this:

Property Value from Environment

```
1 @Service
2 public class IssueServiceBean implements IssueService {
3
4     @Autowired
5     private Environment environment;
6
7     private String defaultTitle = environment.getProperty("issue.title.default");
8
9     // ...
10
11 }
```

Packaging a property file within the jar file provides an excellent mechanism to configure default application configuration values. What if some of those values need to change when the application is deployed to various non-production and production server environments? In the next sections, we discuss a few approaches to override default configuration values.

Property File Load Order

In addition to loading configuration files inside the jar file, Spring Boot loads configuration files outside of the jar file. The property source loader searches first inside the jar file. If a configuration file is found, it loads the properties within the file. Next, the loader searches outside the jar file in specific locations. If a configuration file is found outside the jar file, the properties are loaded. If a property exists in both the internal and external configuration file, the value from the external file overlays that from the internal file. The Spring Boot property source loader searches for configuration files in the following location order:

1. The classpath root
2. The classpath `/config` package
3. The current directory
4. The `/config` sub-directory of the current directory

One way to facilitate environment-specific configuration is to leverage the property source loader order. Place all of the default and local development environment-specific property values within the `application.properties` file located inside the jar file. For each server environment, create a separate `application.properties` file stored somewhere outside of your project structure. Within each of the environment-specific configuration files, include the key-value pairs for only those properties with different values than the default inside of the jar file.

When the application executable jar file is deployed to the host server of a particular environment, the environment-specific configuration file is deployed to the same directory or a `/config` sub-directory. When the jar is executed, the property source loader will first load the configuration file inside the jar and then overlay the default properties with environment-specific values contained in the external configuration file.

There are situations where configuration values need to be supplied outside of a property file. The Spring Boot property source loader supports a variety of configuration locations. The complete list is documented in the Spring Boot reference guide chapter titled [Externalized Configuration](#)⁶⁵. Configuration values obtained from command line arguments are loaded last, meaning that their value overrides any previously loaded.

Suppose a production problem can be fixed by altering a property value. Changing a property file embedded within a jar file may be cumbersome or forbidden by company policy. However, restarting the application supplying a command line property value can resolve the issue with minimal down time.

Command line argument property values are commonly used to override the behavior of the property source loader. Suppose that we want the configuration file to be named `myapp.properties` instead of `application.properties`. The configuration file name may be changed using the property `spring.config.name` supplied via a command line argument like this:

```
$ java -jar myapp.jar --spring.config.name=myapp
```

Any property value may be supplied to the application via command line argument by preceeding it with `--`. For example, the server port may be set by using: `--server.port=9080`

Using Profiles with Property Files

To augment the default property file, `application.properties`, profile specific property files may be created to load additional or override existing configuration values. Spring profiles provide

⁶⁵<http://docs.spring.io/spring-boot/docs/1.2.4.RELEASE/reference/htmlsingle/#boot-features-external-config>

the means to conditionally load sets of resources when the profile is activated. When a Spring Boot application is started with one or more active profiles, the property source loader not only searches for `application.properties`, but also `application-{profileName}.properties` as well. The property source loader searches for profile specific configuration files in the same location hierarchy discussed in the previous section.

We have learned how to use external configuration files to supply environment-specific values to the application. Another approach is to use profiles and include all of the configuration files inside the jar file. Let's create a configuration file for a hypothetical quality assurance, or QA, environment. In the `src/main/resources/config` directory, create a file named `application-qa.properties` and include the values shown below.

`application-qa.properties`

```
1 ##
2 # Embedded Server
3 ##
4 server.port=9081
```

Create an executable jar file by running `mvn package` at a command prompt. Next, type `java -jar target/issuetracker-ws-1.0.0-SNAPSHOT.jar --spring.profiles.active=qa` and press *Enter* to start the application. Near the end of the console log, Apache Tomcat prints the HTTP port.

Console Log

```
1 2014-12-02 14:31:21.815 INFO 20644 --- [           main] s.b.c.e.t.TomcatEmbedd\
2 edServletContainer : Tomcat started on port(s): 9081/http
```

The server has started on port 9081 proving the test successful. Maven correctly packaged both of the configuration files and the application automatically loaded and applied the property values for the qa profile.

By leveraging the power of Spring profiles, developers may continue to rapidly prototype and enhance the application on their local machine using `'mvn spring-boot:run'` with the `application.properties` configuration file. When it is time to deploy to a server, operations engineers lives are kept simple, needing only to deploy a jar file and execute it with the environment name in the `--spring.profiles.active` command line parameter.

Spring Boot Property Reference

Much of Spring Boot's behavior is configuration driven. The Spring Boot reference guide contains an [appendix of application properties](#)⁶⁶ and their default values.

⁶⁶<http://docs.spring.io/spring-boot/docs/1.2.4.RELEASE/reference/htmlsingle/#common-application-properties>



Take Only What You Need

Do not include the property values verbatim from the Spring Boot reference guide appendix into your application. Include only the properties whose value you wish to override.



The Road Ahead

In the next chapter, we will switch from HSQLDB to production grade MySQL and MongoDB database technologies.

Production Databases

Thus far in this book, we have used HSQLDB to store issue information. HSQLDB is an in-memory, relational database that is great for prototyping and unit testing. Eventually projects require persistent, production-grade data stores. This chapter introduces relational, or SQL, and document, or NoSQL, database technologies and demonstrates how to integrate one of each into the web services project.

Relational Databases

Relational databases store information in a tabular format. Each table contains rows of data divided into columns of attributes. Generally, each table describes an entity type and the columns represent the entity attributes.

Each table row utilizes one or more columns to form a unique key. Relationships are formed between table rows by storing the unique key from a row in one table within a column of a second table row.

Relational databases have been the mainstay of information storage since the 1970's. There are dozens of proprietary and open source relational databases available today including, but by no means limited to: HSQLDB, H2, MySQL, PostgreSQL, Oracle, and SQL Server.

Relational databases use Structured Query Language, or SQL, to access and manipulate the data and data structures they house. With the advent of structured storage databases, which do not use SQL, relational databases are sometimes referred to as *SQL* databases and structured storage databases called *NoSQL*.

Working with MySQL

MySQL is one of the most popular and widely used open-source relational database systems. It was created in the mid-1990's by a Swedish company, MySQL AB. In the late 2000's MySQL AB was acquired by Sun Microsystems, which, in turn, was purchased by Oracle.

MySQL is commonly used in web applications and serves as the **M** in the **LAMP**⁶⁷ open source web application technology stack.

⁶⁷[http://en.wikipedia.org/wiki/LAMP_\(software_bundle\)](http://en.wikipedia.org/wiki/LAMP_(software_bundle))

Initial Setup

This book does not describe how to install the MySQL database on a computer. The installation is operating system specific and periodically changes as new releases become available. Refer to the [MySQL Documentation](http://dev.mysql.com/doc/)⁶⁸ for installation instructions.

After installing MySQL, log into the command line client, or MySQL Workbench if you have installed it as well. To prepare the database for the application, we must create a database, create a table for issue data, and create a database user account. Use the SQL statements illustrated below to create a database named `issuetracker` and create the issue table within it.

Create the Database

```
1 CREATE DATABASE IF NOT EXISTS issuetracker;
2 USE issuetracker;
3
4 CREATE TABLE `issue` (
5   `id` bigint(20) NOT NULL AUTO_INCREMENT,
6   `title` varchar(255) DEFAULT NULL,
7   `type` varchar(64) DEFAULT NULL,
8   `priority` varchar(64) DEFAULT NULL,
9   `status` varchar(64) DEFAULT NULL,
10  `description` varchar(2000) DEFAULT NULL,
11  PRIMARY KEY (`id`)
12 ) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

The `CREATE DATABASE` SQL statement conditionally creates the `issuetracker` database if it does not already exist. When authoring SQL scripts, this safeguards against damaging an existing database. The `USE` statement switches the current context to be within the `issuetracker` database. Subsequent SQL statements will be applied to that database unless a different database is explicitly specified.

The `CREATE TABLE` statement defines the `issue` table within the `issuetracker` database. The statement contains definitions for the six database columns. The case of the table and column names is important. The table name must be the lower camel case representation of the entity name to which it maps. In this case, the *Issue* JPA entity in lower camel case is *issue*. The column names must match the entity attribute names precisely.

Next, create a MySQL user account for the Issue Tracker web services application. Using the *root* account or a shared account poses a security risk. Creating a dedicated account for each application that accesses the database allows administrators to manage not only access, but to set operational limits, such as maximum number of queries in a time period. Use the SQL statements depicted below to create a MySQL account named `issueuser` and grant permissions for that user to the `issuetracker` database.

⁶⁸<http://dev.mysql.com/doc/>

Create the Application User

```
1 CREATE USER 'issueuser'@'localhost' IDENTIFIED BY 'issuepass';
2
3 GRANT ALL ON issuetracker.* TO 'issueuser'@'localhost';
```

The `CREATE USER` SQL statement creates a new MySQL account. The syntax `'username'@'hostname'` specifies both the user name for the account and the host machines from which that user may connect to MySQL. Since both the web services and MySQL are running on the same machine, the hostname is set to `localhost`. The account password is specified using the `IDENTIFIED BY` clause.

The `CREATE USER` statement creates a MySQL account; however, the user cannot perform any actions until some permissions are granted. MySQL has a fine-grained permission system, but to keep things simple, we will grant the user all permissions on the `issuetracker` database. The `GRANT` statement is used to define the permissions for a user. The statement gives `ALL` permissions `ON` all data structures within the `issuetracker` database to the user.

MySQL setup for the Issue Tracker application is complete. Let's update the application to switch from `HSQLDB` to `MySQL`.

POM Changes

The Maven `pom.xml` file declares the dependencies for the web services project. Currently the POM contains dependencies for the Spring Boot Starter Data JPA library and the `HSQLDB` driver.

POM with `HSQLDB`

```
<dependencies>

...

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
  <groupId>org.hsqldb</groupId>
  <artifactId>hsqldb</artifactId>
  <scope>runtime</scope>
</dependency>

...

</dependencies>
```

As we discussed early in the book, the Spring Boot Starter POM reconciles the dependency versions for known artifacts. Note that the HSQLDB version is not specified. Spring Boot recognizes the HSQLDB dependency and automatically provides the latest supported version of the driver.

The MySQL driver dependency is also managed by Spring Boot. Let's add it to the POM file.

POM with MySQL

```
<dependencies>

...

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
  <groupId>org.hsqldb</groupId>
  <artifactId>hsqldb</artifactId>
  <scope>runtime</scope>
</dependency>
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <scope>runtime</scope>
</dependency>

...

</dependencies>
```

The version of the MySQL Connector for Java library does not need to be specified. Spring Boot automatically reconciles the latest supported version. Also note that the scope value is `runtime`. The application logic will not directly reference MySQL classes and, therefore, does not require the dependency for compilation.

The HSQLDB dependency may remain within the POM. Both the HSQLDB and the MySQL jar files will be included in the executable jar file. In certain circumstances, it is valuable to use an in-memory database and a persistent database in others. For example, it may be advantageous to use an in-memory database to execute unit tests on a continuous integration server due to the ease of setting up and tearing down the database for each execution of the test suite.

Configuration Changes

During Spring Boot application startup, the configuration module examines classpath dependencies and the application configuration to determine if a `DataSource` should be created and how to configure it. When the Spring Boot Starter Data JPA library is found, Spring Boot searches for database connection configuration.

If there is no explicit database connection configuration and the HSQLDB dependency is on the classpath, Spring Boot creates a `DataSource` that connects to an in-memory HSQLDB instance.

If there are database connection properties in the application configuration, Spring Boot creates a `DataSource` that connects to the database specified by the configuration values. Let's add MySQL configuration values to the `application.properties` file.

application.properties DataSource Configuration

```
1 ##
2 # DataSource
3 ##
4 spring.datasource.url=jdbc:mysql://localhost/issuetracker
5 spring.datasource.username=issueuser
6 spring.datasource.password=issuepass
7 spring.datasource.driverClassName=com.mysql.jdbc.Driver
```

Spring Boot automatically configures a connection to the MySQL database using the property values. Since the project uses Spring Data JPA, the `DataSource` will automatically provide connection pooling features.



Datasource Property Reference

Spring Boot creates pooling `DataSource` when used in conjunction with the Spring Data JPA library. Spring Boot configures the connection pool with sensible defaults. To override the default values you may declare additional properties from the [Spring Boot property reference](http://docs.spring.io/spring-boot/docs/1.2.4.RELEASE/reference/htmlsingle/#common-application-properties)⁶⁹.

There are no code changes required to use MySQL in lieu of HSQLDB. The Spring Data JPA project abstracts implementation details of the underlying database from the application code. The next time the web service project is started, it will connect to MySQL.

Running the Application

Let's test the new functionality. Open a terminal window and navigate to the web services project base directory. Type `mvn spring-boot:run` and press *Enter* to start the web services on port 8080.

⁶⁹<http://docs.spring.io/spring-boot/docs/1.2.4.RELEASE/reference/htmlsingle/#common-application-properties>

Open a second terminal window, or tab, and navigate to the user interface project base directory. Type `grunt run` and press *Enter* to start the local web server on port 9000. Open a third terminal window and access the MySQL command line interface or open MySQL Workbench.

Open a browser and type `http://localhost:9000/index.html` into the address bar and press *Enter*. The issue manager module displays an empty issue list. Click the add issue button in the top navigation bar and create a new issue. Query the issue table with `select * from issue;`. The issue table contains a row of data for the issue created.

Issue Table

```
mysql> select * from issue;
```

id	title	type	priority	status	description
1	Improve It	ENHANCEMENT	LOW	OPEN	Let's change this ...

```
1 row in set (0.00 sec)
```

Using the issue tracker user interface, perform a variety of activities creating, editing, and deleting issues. Periodically query the database to see the changes take effect.

HSQldb stores issues in the memory space of the Java process running the web services. When the application is restarted, the HSQldb instance is restarted as well, losing the issue data. That is not the case with MySQL. MySQL writes the data to disk and caches it in its own memory space. Restart the web services application and navigate to the issue list in the user interface. The list of issues contains the issue data with the state unchanged.

NoSQL Databases

NoSQL databases store information in a format different than the tabular format used by relational databases. NoSQL databases derive their name from the fact that they do not use Structured Query Language, or SQL, to perform database operations.

Document databases are a sub-category of NoSQL databases that serialize entity data into a structured data format known generically as a *document*. Most document databases use XML, YAML, JSON, or BSON document formats. Entities are serialized, or transformed, into documents and stored within the database.

In document databases, many entity relationships are captured within a single document. The document represents a complete entity graph. For example, suppose there are two entities: Person and Address. An instance of Person may have many addresses such as *home*, *work*, etc. Within a relational database, the information would be stored in two tables with a foreign key relationship. In

a document database, the information is stored in a single collection with the document data format describing the *person has one-to-many addresses* relationship.

NoSQL databases emerged in the 2000's. There are several proprietary and open source NoSQL databases available today including, but by no means limited to: MongoDB, Couchbase, CouchDB, Cassandra, Dynamo, and Redis.

Working with MongoDB

MongoDB is one of the most popular NoSQL database systems. It was created in the mid-2000's by the software company 10gen. In 2009, the project became open source software with 10gen, now MongoDB Inc., offering commercial support.



Git Repository for MongoDB

The issue tracker web services project has a dedicated repository for MongoDB persistence. While very little of the application code is different, the preferred way to illustrate the best practices for using the Spring Data MongoDB project is through a separate repository.

Git Repository: [issue-tracker-ws-boot-mongodb](https://github.com/mwarman/issue-tracker-ws-boot-mongodb)⁷⁰

Initial Setup

This book does not describe how to install MongoDB on a computer. The installation is operating system specific and periodically changes as new releases become available. Refer to the [MongoDB Documentation](http://docs.mongodb.org/manual/installation/)⁷¹ for installation instructions.

After installing MongoDB, log into the command line client. Open a terminal window and type `mongo`.

To prepare the database for the application, we must create a database, create a collection for issue data, and create a database user account. Use the commands illustrated below to create a database named `issuetracker`.

Create the Database

```
1 > use issuetracker
2 switched to db issuetracker
```

That's it! MongoDB detects that you have requested to use a database that doesn't exist. It creates the database for you. To view all databases, use the `show dbs` command.

⁷⁰<https://github.com/mwarman/issue-tracker-ws-boot-mongodb>

⁷¹<http://docs.mongodb.org/manual/installation/>

List the Databases

```

1 > show dbs
2 issuetracker      0.203125GB
3 local             0.078125GB
4 test              0.203125GB

```

Next, create a MongoDB user account for the Issue Tracker web services application. Creating a dedicated account for each application that accesses the database allows administrators to manage access and set operational limits, such as maximum number of connections. Use the command depicted below to create a MongoDB account named `issueuser`.

Create the Application User

```

1 > use issuetracker
2 switched to db issuetracker
3
4 > db.addUser(
5   ... { user: "issueuser",
6   ...   pwd: "issuepass",
7   ...   roles: [ "readWrite" ]
8   ... }
9   ... )
10
11 {
12     "user" : "issueuser",
13     "pwd"  : "9ba6f3a3fac04644cf68a31cf5d33b09",
14     "roles" : [
15         "readWrite"
16     ],
17     "_id"  : ObjectId("548741576758cb0ac81fe127")
18 }

```

First, ensure that the account is created within the `issuetracker` database by issuing the `use issuetracker` command.

Then execute the `db.addUser` command to create a user account for the application with the [built-in role](http://docs.mongodb.org/manual/reference/built-in-roles/)⁷² `readWrite`. Note that the command did not specify which database the `issueuser` is granted permission to access. The MongoDB authorization system is different than MySQL. Each database has a system collection containing the users granted access to it. The `db.` prefix to the `addUser` command instructs MongoDB to create the user in the database you are currently using. That is why the `use issuetracker` command is issued before adding the user.

⁷²<http://docs.mongodb.org/manual/reference/built-in-roles/>



MongoDB Versions

The `addUser` command depicted above is used in MongoDB version 2.4. In version 2.6, the command is renamed `createUser` and the data structure within the command body changed. At a terminal prompt, type `mongo --version` to see which version of MongoDB is installed. See the [add user page](http://docs.mongodb.org/manual/tutorial/add-user-to-database/)⁷³ of the MongoDB documentation for assistance. Use the *Options* in the lower-left corner of the documentation to find the documentation for your version of MongoDB.

MongoDB setup for the Issue Tracker application is complete. Let's update the application to use it.

POM Changes

The Maven `pom.xml` file declares the dependencies for the web services project. The POM we've used throughout this book contains dependencies for the Spring Boot Starter Data JPA library and the HSQLDB driver. Earlier in this chapter, we added the MySQL driver alongside the HSQLDB driver dependency. In the MongoDB project, the HSQLDB, MySQL, and Spring Data JPA dependencies are replaced. To integrate with MongoDB, include the Spring Data MongoDB starter POM dependency.

POM with MongoDB

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
<groupId>org.hsqldb</groupId>
<artifactId>hsqldb</artifactId>
<scope>runtime</scope>
</dependency>
<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
<scope>runtime</scope>
</dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
```

⁷³<http://docs.mongodb.org/manual/tutorial/add-user-to-database/>

```

    <artifactId>spring-boot-starter-data-mongodb</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>

```

The snippet above depicts the dependencies section of the POM file. The relational database dependencies have been removed and the MongoDB dependency added.

Configuration Changes

During Spring Boot application startup, the configuration module examines classpath dependencies and the application configuration to determine if a database connection should be created and how to configure it. When the Spring Boot Starter Data MongoDB library is found, Spring Boot searches for database connection configuration.

If there are database connection properties in the application configuration, Spring Boot creates a connection to the MongoDB database specified by the configuration values. If explicit configuration is omitted, the Spring Data MongoDB project attempts to create a connection to `mongodb://localhost/test`. Let's add the necessary configuration values to the `application.properties` file.

Datasource Configuration

```

1  ##
2  # Datasource
3  ##
4  spring.data.mongodb.uri=mongodb://issueuser:issuepass@localhost:27017/issuetrack\
5  er
6
7  ##
8  # Data
9  ##
10 spring.dao.exceptiontranslation.enabled=false

```

Spring automatically configures a connection to the MongoDB database using the property values. There is no explicit Java configuration required.

The Spring Data MongoDB project abstracts the implementation details of the underlying database from the application code. However, there are a few differences to the application when using a NoSQL database instead of a JPA managed relational database.

Entity Changes

The application changes necessary to use MongoDB instead of a relational database are very slight. Primarily, the changes are within the model class. To persist an Issue in MongoDB open the class and make the changes illustrated below.

Issue Entity for MongoDB

```
1  @Document
2  public class Issue {
3
4      @Id
5      private String id;
6
7      private String title;
8
9      private String description;
10
11     private IssueType type;
12
13     private IssuePriority priority;
14
15     private IssueStatus status;
16
17     public Issue() {
18
19     }
20
21     // ....
22
23 }
```

Rather than annotating entity classes with `@Entity`, the class is annotated with `@Document`. Spring Data scans the project when the application starts and identifies all persistent `@Document` classes.

The `id` attribute is still annotated with `@Id` however, this annotation is from the `org.springframework.data.annotation` package rather than the `javax.persistence` package. The annotation serves the same purpose. It denotes the primary key identifier for the document.

When using MongoDB, the primary key attribute should be declared as either a `String` or a `BigInteger`. The Issue is updated to declare the `id` attribute to be a `String` type.

Finally, the `@Enumerated` annotations have been omitted from the `type`, `priority`, and `status` attributes. The annotation is specific to JPA persistence and instructs the Spring Data JPA project how to map an enum type for storage in the database. The Spring Data MongoDB package maps enum values using the their code value.

Repository Changes

The repository interface is typed with the entity class and primary key identifier attribute type. Since we use a `String` primary key identifier in the `Issue` document class, the repository must be changed accordingly.

The JPA repository interface extends `JpaRepository`. The repository interface for MongoDB persistence extends the `PagingAndSortingRepository` interface. Open the `IssueRepository` class and update it as depicted below.

IssueRepository for MongoDB

```
1 @Repository
2 public interface IssueRepository extends
3     PagingAndSortingRepository<Issue, String> {
4
5 }
```

Service Changes

The service interface and implementation must also be changed. Open the `IssueService` and `IssueServiceBean` source files and change the `find` and `delete` methods to use `String` instead of `Long` as the primary key identifier type. Change the return type of the `findAll` method from `List<Issue>` to `Iterable<Issue>` to accommodate the return type from `PagingAndSortingRepository`.

IssueService for MongoDB

```
1 public interface IssueService {
2
3     /**
4      * Search the issue data repository for all Issue entities.
5      * @return An Iterable collection of Issue entities or null if none found.
6      */
7     Iterable<Issue> findAll();
8
9     /**
10      * Search the issue data repository for a single Issue entity by the primary
11      * key identifier.
12      * @param id An issue primary key identifier.
13      * @return An Issue entity or null if not found.
14      */
15     Issue find(String id);
16
17     // ...
```

```
18
19  /**
20   * Delete an Issue entity from the data repository.
21   * @param id The primary key identifier of the issue to delete.
22   */
23  void delete(String id);
24
25 }
```

Within the service implementation, update the `findAll` method to use `Iterable` and change the type of the `find` and `delete` method parameter to a `String`.

IssueServiceBean for MongoDB

```
1  @Override
2  public Iterable<Issue> findAll() {
3      logger.info("> findAll");
4
5      Iterable<Issue> issues = issueRepository.findAll();
6
7      logger.info("< findAll");
8      return issues;
9  }
10
11  @Override
12  public Issue find(String id) {
13      logger.info("> find id:{", id);
14
15      Issue issue = issueRepository.findOne(id);
16
17      logger.info("< find id:{", id);
18      return issue;
19  }
20
21  // ...
22
23  @Override
24  public void delete(String id) {
25      logger.info("> delete");
26
27      issueRepository.delete(id);
28
29      logger.info("< delete");
```

```

30     }
31
32     // ....

```

Controller Changes

Finally, we must also change the request mapping methods of the web service controller class. Enhance the `getAllIssues` method to use `Iterable` instead of `List`. Change the `id` method parameters to `String` on the `getIssue` and `deleteIssue` methods.

IssueController for MongoDB

```

1     @RequestMapping(
2         value = "/issues",
3         method = RequestMethod.GET,
4         produces = MediaType.APPLICATION_JSON_VALUE)
5     public ResponseEntity<Iterable<Issue>> getAllIssues() {
6         logger.info("> getAllIssues");
7
8         Iterable<Issue> issues = null;
9         try {
10             issues = issueService.findAll();
11
12             // if no issues found, create an empty list
13             if (issues == null) {
14                 issues = new ArrayList<Issue>();
15             }
16         } catch (Exception e) {
17             logger.error("Unexpected Exception caught.", e);
18             return new ResponseEntity<Iterable<Issue>>(issues,
19                 HttpStatus.INTERNAL_SERVER_ERROR);
20         }
21
22         logger.info("< getAllIssues");
23         return new ResponseEntity<Iterable<Issue>>(issues, HttpStatus.OK);
24     }
25
26     @RequestMapping(
27         value = "/issues/{id}",
28         method = RequestMethod.GET,
29         produces = MediaType.APPLICATION_JSON_VALUE)
30     public ResponseEntity<Issue> getIssue(@PathVariable("id") String id) {
31         logger.info("> getIssue");

```

```
32
33     Issue issue = null;
34     try {
35         issue = issueService.find(id);
36
37         // if no issue found, return 404 status code
38         if (issue == null) {
39             return new ResponseEntity<Issue>(HttpStatus.NOT_FOUND);
40         }
41     } catch (Exception e) {
42         logger.error("Unexpected Exception caught.", e);
43         return new ResponseEntity<Issue>(issue,
44             HttpStatus.INTERNAL_SERVER_ERROR);
45     }
46
47     logger.info("< getIssue");
48     return new ResponseEntity<Issue>(issue, HttpStatus.OK);
49 }
50
51 // ...
52
53 @RequestMapping(
54     value = "/issues/{id}",
55     method = RequestMethod.DELETE)
56 public ResponseEntity<Issue> deleteIssue(@PathVariable("id") String issueId)\
57 {
58     logger.info("> deleteIssue");
59
60     try {
61         issueService.delete(issueId);
62     } catch (Exception e) {
63         logger.error("Unexpected Exception caught.", e);
64         return new ResponseEntity<Issue>(HttpStatus.INTERNAL_SERVER_ERROR);
65     }
66
67     logger.info("< deleteIssue");
68     return new ResponseEntity<Issue>(HttpStatus.NO_CONTENT);
69 }
70
71 // ...
```

In the request mapping methods the parameter annotated with `@PathVariable` must be changed

from Long to String. This allows the value of the `id` attribute to be mapped to the correct data type when the web service request is received. It can be passed to the service and the repository as a String.

Running the Application

Let's test the new functionality. Open a terminal window and navigate to the web services project base directory. Type `mvn spring-boot:run` and press *Enter* to start the web services on port 8080. Open a second terminal window, or tab, and navigate to the user interface project base directory. Type `grunt run` and press *Enter* to start the local web server on port 9000. Open a third terminal window and access the MongoDB command line interface by typing `mongo issuetracker`.

Open a browser and type `http://localhost:9000/index.html` into the address bar and press *Enter*. The issue manager module displays an empty issue list. Click the add issue button in the top navigation bar and create a new issue. Query the issue table with `db.issue.find()`. The issue collection contains a row of data for the issue created. To count the number of documents in the issue collection, use the MongoDB command `db.issue.count()`.

Issue Collection

```
user@ubuntu:~$ mongo issuetracker
```

```
MongoDB shell version: 2.4.12
```

```
connecting to: issuetracker
```

```
> db.issue.find()
{ "_id" : ObjectId("548747c7e4b0d0c745462dbb"), "_class" : "org.example.issuetra\
cker.model.Issue", "title" : "My Task", "description" : "This is my task.", "typ\
e" : "TASK", "priority" : "MEDIUM", "status" : "IN_PROGRESS" }
{ "_id" : ObjectId("54874813e4b0d0c745462dbd"), "_class" : "org.example.issuetra\
cker.model.Issue", "title" : "Important Change", "description" : "This is a high\
priority change.", "type" : "ENHANCEMENT", "priority" : "HIGH", "status" : "OPE\
N" }
{ "_id" : ObjectId("5487481fe4b0d0c745462dbe"), "_class" : "org.example.issuetra\
cker.model.Issue", "title" : "Small Problem", "description" : "We have a small p\
roblem here.", "type" : "BUG", "priority" : "LOW", "status" : "DONE" }
{ "_id" : ObjectId("548827a5e4b072e323e91121"), "_class" : "org.example.issuetra\
cker.model.Issue", "title" : "New Feature", "description" : "This is a user stor\
y.", "type" : "STORY", "priority" : "LOW", "status" : "OPEN" }

> db.issue.count()
4

>
```

Using the issue tracker user interface, perform a variety of activities creating, editing, and deleting issues. Periodically query the database to see the changes take effect.



The Road Ahead

In the next chapter, we will discuss the out-of-the-box production operations features of Spring Boot Actuator.

Production Monitoring and Management

Production applications are routinely monitored to ensure they are operational and error free. Application throughput statistics are required to determine load trends. When problems arise, operations engineers require the ability to trace application processes as a first triage step. These operational features typically require that software engineers create custom solutions, generating additional application development costs and solutions to maintain.

Introducing Actuator

The Spring Boot Actuator project provides out-of-the-box production operations features to monitor and manage applications. Actuator makes these features available through HTTP, JMS, or a remote SSH or Telnet shell. Due to the sensitive nature of operational capabilities and information, Actuator is fully integrated with the Spring Security project. Features may be enabled, disabled, or secured through configuration or customized with a small amount of code.

Enabling Actuator

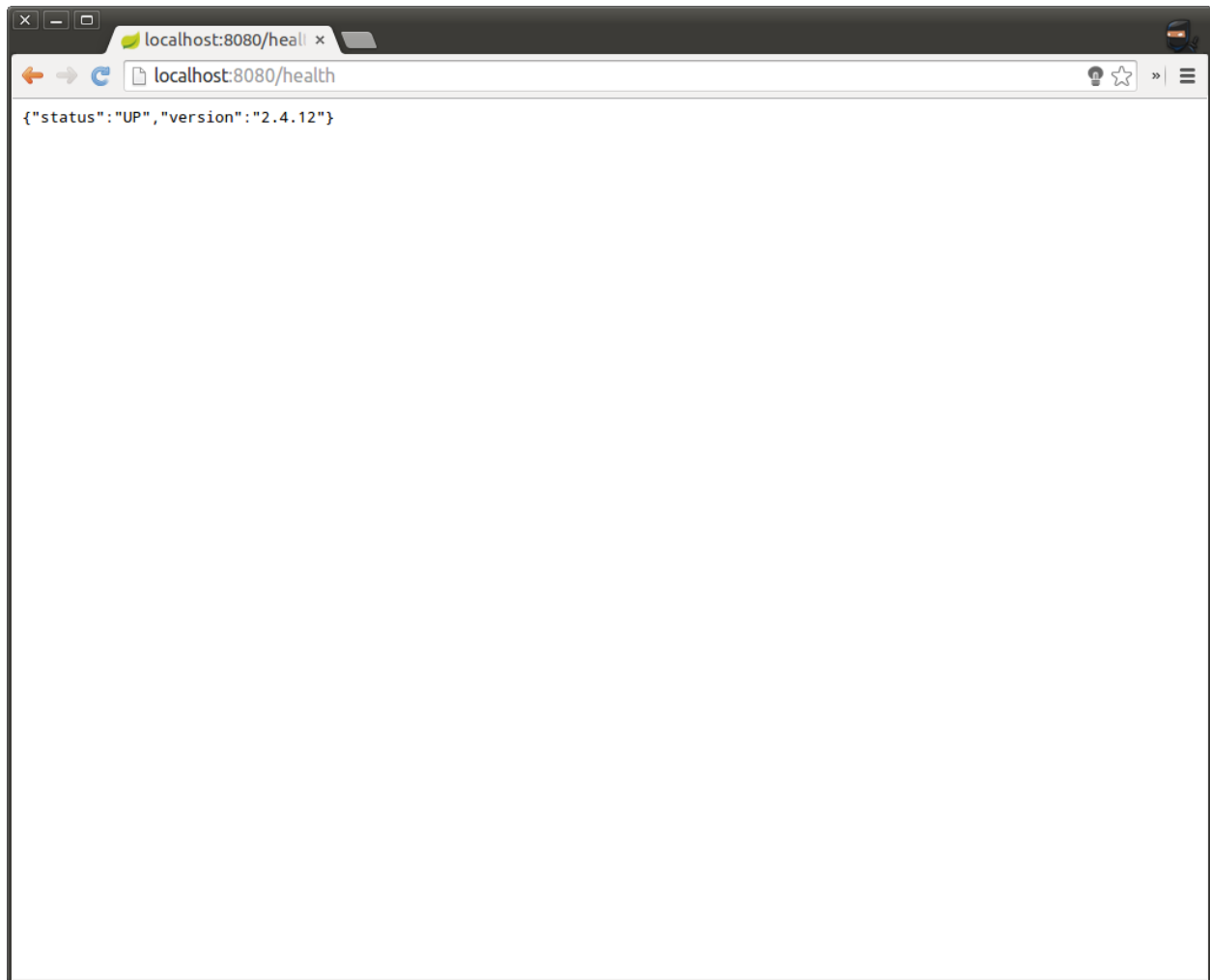
To enable Actuator capabilities within the web services project, simply include the dependency in the POM file. The Spring Boot Actuator starter POM is bootstrapped with sensible default configurations. Open the `pom.xml` file and include the Actuator dependency as illustrated below.

Actuator Maven Dependency

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

When the Spring Boot application starts it detects Actuator and automatically enables its monitoring and management endpoints. Since the web services project also includes the Spring Boot starter web dependency, Actuator exposes the endpoints over HTTP.

To see Actuator in action, start the web services application and open a browser to `http://localhost:8080/health`. The HTTP context path `/health` is the default Actuator health check HTTP endpoint.



Actuator - Health Check Endpoint

Endpoints

The Actuator project provides several out-of-the-box endpoints and the framework to construct custom endpoints. In the previous section, we demonstrated the health check capability exposed via HTTP at `/health`. Each Actuator endpoint is mapped to a context path that matches its ID.

The table below describes the Actuator features highlighted in this book. For a full list of Actuator capabilities, see the [Actuator reference guide](http://docs.spring.io/spring-boot/docs/1.2.4.RELEASE/reference/htmlsingle/#production-ready-endpoints)⁷⁴.

⁷⁴<http://docs.spring.io/spring-boot/docs/1.2.4.RELEASE/reference/htmlsingle/#production-ready-endpoints>

Featured Actuator Endpoints

ID	Description
health	Provides application health information.
info	Provides application descriptive information.
metrics	Provides status information.
trace	Provides information about the last few requests.



Security

When Spring Security is on the application classpath, Actuator automatically adds basic authentication to all endpoints except health and info.

Configuration

Endpoints

Actuator endpoints may be configured using the application property file. The property key for each endpoint follows the format: `endpoints + . + ID + . + Feature`. Properties are used to configure if an endpoint is enabled, if it should be considered sensitive, and the endpoint name.

For example, to customize the health endpoint, add the following to the `application.properties` file.

Sample Actuator Endpoint Customization

```
endpoints.health.id=status
endpoints.health.sensitive=true
endpoints.health.enabled=true
```

The configuration depicted above, changes the ID of the health endpoint to *status*, making the HTTP context path `/status`. The sensitive attribute is set to `true` requiring authentication to access it. The enabled property is set to `true`, exposing the health capability. To disable the health endpoint entirely, set the enabled property to `false`.

HTTP

To group the Actuator endpoints under a common HTTP context path, use the `management.context-path` property. Open the `application.properties` file and include the configuration shown below to set the context path for Actuator endpoints.

Actuator HTTP Context Path Configuration

```
management.context-path=/actuators
```

If Spring Security is on the classpath, all endpoints classified sensitive are automatically protected by basic authentication. Our web services project does not include Spring Security; however, if it did, the user name, password, and role could be customized using the following property values.

Actuator HTTP Security Configuration

```
security.user.name=prodops  
security.user.password=s3cur!ty  
management.security.role=OPERATIONS
```

To disable HTTP endpoints completely, set the `management.port` property value to `-1`.

Disable HTTP Actuator Endpoints

```
management.port=-1
```

Health Check

The Actuator health check feature evaluates the basic status of application components. If it detects that a `DataSource` object is present, or a Redis, MongoDB, or RabbitMQ connection, Actuator automatically performs basic verification that those dependencies are functioning properly.

Start the web services application. Open a browser and navigate to `http://localhost:8080/actuators/health` (or just `/health` if you have not set the `management.context-path` property). The health check response depicted below indicates the application is healthy with the status value “UP”. Note that MySQL was verified as a part of determining the overall application status.

Successful Health Check

```
{  
  "status": "UP",  
  "database": "MySQL",  
  "hello": 1  
}
```

Now, let's stop the MySQL service and check the health endpoint again. The application status is now “DOWN”. The error message indicates the reason for the application failure. Not surprisingly, failure to communicate with MySQL is the root cause of the problem.

Failing Health Check

```
{
  "status": "DOWN",
  "database": "MySQL",
  "error": "org.springframework.dao.RecoverableDataAccessException: StatementCall\
back; SQL [SELECT 1]; Communications link failure\n\nThe last packet successful\
ly received from the server was 39,306 milliseconds ago. The last packet sent s\
uccessfully to the server was 16 milliseconds ago.; nested exception is com.mysq\
l.jdbc.exceptions.jdbc4.CommunicationsException: Communications link failure\n\n\
The last packet successfully received from the server was 39,306 milliseconds ag\
o. The last packet sent successfully to the server was 16 milliseconds ago."
}
```

**Restart the Application**

After restarting MySQL, you will need to restart the web services to reestablish database connectivity.

Creating Custom Health Checks

To add custom health checks to be executed when the Actuator health endpoint is invoked, Spring provides the `HealthIndicator` interface. Simply create an implementation of `HealthIndicator` and register with the Spring application context by adding the class-level annotation `@Component`.

Example HealthIndicator Implementation

```
1  @Component
2  public class ExampleHealthIndicator implements HealthIndicator {
3
4      @Override
5      public Health health() {
6
7          try {
8              // Execute health checks
9              // ...
10             return new Health.Builder().up().withDetail("attribute", "value").build();
11         } catch (Exception e) {
12             return new Health.Builder().down(e).build();
13         }
14     }
```

```
15     }  
16  
17 }
```

Application Information

The Actuator info endpoint returns application descriptive information. The endpoint behavior returns an empty JSON object; however, this may be easily customized through the configuration file.

Creating Custom Application Information

To add key value pairs to the info endpoint response, Actuator reads the application configuration file searching for property keys prefixed with `info..` Open the `application.properties` file and include the following values:

Actuator Info Endpoint Custom Configuration

```
info.app.name=IssueTrackerWS  
info.app.description=Issue Tracker Web Services  
info.app.version=1.0.0
```

After restarting the application, open a browser and navigate to `http://localhost:8080/actuators/info` to see the values in the info endpoint response.

Info Response

```
{  
  "app": {  
    "description": "Issue Tracker Web Services",  
    "version": "1.0.0",  
    "name": "IssueTrackerWS"  
  }  
}
```

Metrics

The Actuator metrics endpoint provides status and key performance indicator values for the application components. The *counter* and *gauge* attributes measure application utilization. The *heap*, *threads*, *classes*, and *gc* attributes describe the status of the JVM in which the application is running.

The *mem* and *processors* attributes provide insight into the host machine operational state. The *uptime* attributes indicate the number of milliseconds since the application started.

To see a meaningful response from the metrics service, start the web services and the user interface. Using a browser, create, edit, and delete several issues to create data for the counter and gauge metrics. Then navigate to <http://localhost:8080/actuators/metrics> to see the information provided by the metrics endpoint.

Metrics Response

```
{
  "counter.status.200.actuators.info":1,
  "counter.status.200.actuators.metrics":1,
  "counter.status.200.issues":3,
  "counter.status.200.issues.11":1,
  "counter.status.200.issues.12":1,
  "counter.status.200.issues.3":1,
  "counter.status.200.issues.id":3,
  "counter.status.201.issues":1,
  "counter.status.204.issues.id":1,
  "gauge.response.actuators.info":39.0,
  "gauge.response.actuators.metrics":22.0,
  "gauge.response.issues":7.0,
  "gauge.response.issues.11":2.0,
  "gauge.response.issues.12":1.0,
  "gauge.response.issues.3":1.0,
  "gauge.response.issues.id":17.0,
  "mem":465408,
  "mem.free":378906,
  "processors":4,
  "uptime":472576,
  "instance.uptime":467021,
  "heap.committed":465408,
  "heap.init":385470,
  "heap.used":86501,
  "heap":5483520,
  "threads.peak":22,
  "threads.daemon":20,
  "threads":22,
  "classes":8113,
  "classes.loaded":8113,
  "classes.unloaded":0,
  "gc.ps_scavenge.count":7,
  "gc.ps_scavenge.time":95,
```

```
"gc.ps_marksweep.count":0,  
"gc.ps_marksweep.time":0  
}
```

The sample metrics response above provides a wealth of operational values that may be used to create an operational dashboard user interface. The counter attributes indicate that the `/issues` web service has returned HTTP 200 response 3 times, HTTP 201 1 time, and HTTP 204 1 time. This means the `getAllIssues` service responded correctly 3 times, 1 issue was created successfully, and 1 deleted successfully.

The gauge attributes indicate the last response time, in milliseconds, for each web service called since the application started.

Creating Custom Metrics

To add custom *counter* or *gauge* attributes to the metrics response, inject a `CounterService` or `GaugeService` into an application component. Suppose we wish to collect counter metrics for `IssueService` method invocations. Simply inject an instance of `CounterService` into the `IssueServiceBean` then use the `counterService.increment` method to gather statistics for each `IssueService` method.

Example Custom Counter Implementation

```
1  @Service  
2  public class IssueServiceBean implements IssueService {  
3  
4      private Logger logger = LoggerFactory.getLogger(this.getClass());  
5  
6      @Autowired  
7      private IssueRepository issueRepository;  
8  
9      @Autowired  
10     private CounterService counterService;  
11  
12     @Override  
13     public List<Issue> findAll() {  
14         logger.info("> findAll");  
15  
16         counterService.increment("services.issueservice.findAll.invoked");  
17  
18         List<Issue> issues = issueRepository.findAll();  
19  
20         logger.info("< findAll");  
21         return issues;  
22     }  
23 }
```

```

22     }
23
24     // ...
25
26 }

```

By including the Spring Boot Actuator starter POM, Actuator components, like `CounterService`, are automatically created and added to the application context when the application is started. In the snippet above, an instance of `CounterService` is injected into the `IssueServiceBean`. Within each of the `IssueServiceBean` methods, the `increment` method is invoked passing the name of the counter.

After restarting the web services and performing several activities with the Issue Tracker user interface, navigate to the metrics endpoint at <http://localhost:8080/actuators/metrics>. The response now contains the custom counter `.service` attributes measuring the load on the `IssueServiceBean`.

Metrics Response with Custom Counters

```

{
  "counter.services.issueservice.create.invoked":1,
  "counter.services.issueservice.delete.invoked":1,
  "counter.services.issueservice.findAll.invoked":2,
  "counter.services.issueservice.update.invoked":2,
  "counter.status.200.actuators.metrics":2,
  "counter.status.200.issues":3,
  "counter.status.200.issues.1":1,
  "counter.status.200.issues.12":1,
  "counter.status.200.issues.id":2,
  "counter.status.201.issues":1,
  "counter.status.204.issues.id":1,
  "gauge.response.actuators.metrics":6.0,
  "gauge.response.issues":8.0,
  "gauge.response.issues.1":3.0,
  "gauge.response.issues.12":1.0,
  "gauge.response.issues.id":27.0,
  "mem":465408,
  "mem.free":371338,
  "processors":4,
  "uptime":118889,
  "instance.uptime":113296,
  "heap.committed":465408,
  "heap.init":385470,
  "heap.used":94069,
  "heap":5483520,

```

```

    "threads.peak":22,
    "threads.daemon":20,
    "threads":22,
    "classes":8112,
    "classes.loaded":8112,
    "classes.unloaded":0,
    "gc.ps_scavenge.count":7,
    "gc.ps_scavenge.time":112,
    "gc.ps_marksweep.count":0,
    "gc.ps_marksweep.time":0
  }

```

Tracing

Inevitably problems arise with production applications. The Actuator trace endpoint is automatically enabled when the Spring Boot Starter Web dependency is included in a project. The trace endpoint provides critical troubleshooting information for the most recent HTTP requests to the application. An example trace response is shown below.

Tracing Response

```

[
  {
    "timestamp":1418227276869,
    "info":{
      "method":"POST",
      "path":"/issues",
      "headers":{
        "request":{
          "host":"localhost:8080",
          "connection":"keep-alive",
          "content-length":"140",
          "accept":"application/json, text/javascript, */*; q=0.01",
          "origin":"http://localhost:9000",
          "user-agent":"Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 \
(KHTML, like Gecko) Chrome/39.0.2171.71 Safari/537.36",
          "content-type":"application/json",
          "referer":"http://localhost:9000/index.html",
          "accept-encoding":"gzip, deflate",
          "accept-language":"en-US,en;q=0.8"
        },
        "response":{

```

```

        "Access-Control-Allow-Origin": "*",
        "Access-Control-Allow-Methods": "DELETE, GET, OPTIONS, PATCH, POST\
, PUT",
        "Access-Control-Max-Age": "3600",
        "Access-Control-Allow-Headers": "x-requested-with, content-type",
        "X-Application-Context": "application:",
        "Content-Type": "application/json",
        "Transfer-Encoding": "chunked",
        "Date": "Wed, 10 Dec 2014 16:01:16 GMT",
        "status": "201"
    }
}
},
{
    "timestamp": 1418227279530,
    "info": {
        "method": "GET",
        "path": "/issues",
        "headers": {
            "request": {
                "host": "localhost:8080",
                "connection": "keep-alive",
                "accept": "application/json, text/javascript, */*; q=0.01",
                "origin": "http://localhost:9000",
                "user-agent": "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 \
(KHTML, like Gecko) Chrome/39.0.2171.71 Safari/537.36",
                "referrer": "http://localhost:9000/index.html",
                "accept-encoding": "gzip, deflate, sdch",
                "accept-language": "en-US,en;q=0.8"
            },
            "response": {
                "Access-Control-Allow-Origin": "*",
                "Access-Control-Allow-Methods": "DELETE, GET, OPTIONS, PATCH, POST\
, PUT",
                "Access-Control-Max-Age": "3600",
                "Access-Control-Allow-Headers": "x-requested-with, content-type",
                "X-Application-Context": "application:",
                "Content-Type": "application/json",
                "Transfer-Encoding": "chunked",
                "Date": "Wed, 10 Dec 2014 16:01:19 GMT",
                "status": "200"
            }
        }
    }
}

```

```
}  
}  
}  
}  
]
```

Conclusions

This chapter provides an overview of the production monitoring and management features offered by Spring Boot Actuator. To learn more about the Actuator suite of capabilities, see the [reference guide](http://docs.spring.io/spring-boot/docs/1.2.4.RELEASE/reference/htmlsingle/#production-ready)⁷⁵.

Actuator offers a rich suite of out-of-the-box production operations features which save development teams the expense of creating and maintaining custom functionality. The capabilities are configuration driven, allowing teams to decide when to enable features and how to expose and secure them. As applications upgrade to the latest release of the Spring Boot platform, Actuator is automatically upgraded as well.

⁷⁵<http://docs.spring.io/spring-boot/docs/1.2.4.RELEASE/reference/htmlsingle/#production-ready>

Part VI. Epilogue

Congratulations! Whether you have just read the book or created the application projects as well, take a moment to reward yourself for completing this journey. Taking command of your professional improvement and personal growth will reward you again and again throughout your career.

“Make the most of yourself, for that is all there is of you.”

– Ralph Waldo Emerson

Part VII. Appendices

Appendix A: Dependency Versions

The issue tracker projects described in this book use several open source, third party dependencies. The dependency versions used in this revision of the book are documented below.

Issue Tracker Web Services

- Spring Boot v1.2.4

Issue Tracker User Interface

- Bootstrap v3.3.5
- Font Awesome v4.3.0
- Backbone.Marionette v2.4.2
- Backbone.Syphon v0.5.0
- Backbone v1.2.1
- Underscore v1.8.3
- jQuery v2.1.4
- log4javascript v1.4.13
- json2



Backbone.Marionette Dependency

The Backbone.Marionette JavaScript used in this project is the minified version of the distribution which also contains the Backbone.Babysitter and Backbone.Wreqr projects. There is also a distribution named *core* which omits these two dependencies. Marionette requires these dependencies so we will include the distribution file which packages them with the Backbone.Marionette JavaScript.



json2.js Dependency

The json2.js dependency does not have formal versions. We include the json2.js file which is shipped with the Backbone.Marionette distribution.

Appendix B: Iconography

Font Awesome icons are employed throughout the user interface application. This appendix describes the icons used and their purpose.

Issue Type Icons

Icon	Purpose
fa-bug	Bug issue type.
fa-check	Task issue type.
fa-book	Story issue type.
fa-wrench	Enhancement issue type.

Issue Priority Icons

Icon	Purpose
fa-arrow-down	Low priority.
fa-chevron-up	Medium priority.
fa-arrow-up	High priority.

Miscellaneous Icons

Icon	Purpose
fa-list	List entities.
fa-plus	Add or create an entity.
fa-trash	Delete an entity.
fa-pencil	Edit an entity.
fa-circle-o-notch	Spinner.

Appendix C: CoffeeScript

CoffeeScript⁷⁶ is a programming language that compiles into JavaScript. JavaScript libraries may be referenced within CoffeeScript source files and vice versa. The command line version of CoffeeScript, `coffee`, is installed using the node package manager. To install CoffeeScript with `npm` use the following command.

```
$ npm install -g coffee-script
```

Many software engineers prefer CoffeeScript syntax to JavaScript. Due to its popularity, the Issue Tracker SPA project is also available on GitHub written entirely in CoffeeScript.



GitHub

The [issue-tracker-ui-marionette-coffee](https://github.com/mwarman/issue-tracker-ui-marionette-coffee)⁷⁷ repository contains the complete Single Page Application user interface project authored in CoffeeScript.

⁷⁶<http://coffeescript.org/>

⁷⁷<https://github.com/mwarman/issue-tracker-ui-marionette-coffee>

Appendix D: Gulp

Gulp⁷⁸ is a build and assembly workflow automation alternative to Grunt. Gulp uses the Node.js I/O streaming capability to execute build tasks in parallel, resulting in faster builds.

The Gulp project has a large plugin ecosystem. Most, if not all, of the plugins available for Grunt are also available for Gulp with nearly identical configuration options.

The Gulp control file is authored in JavaScript (or CoffeeScript). Tasks are defined as functions which typically process a portion of the project, preparing it for local development testing or distribution to a server.

The Gulp project is relatively new compared to Grunt. Many developers are migrating to Gulp or experimenting with its capabilities. The Issue Tracker SPA user interface project has a Gulp control file in addition to Grunt. The Gulp control file, `gulpfile.js` or `gulpfile.coffee`, contains identical functionality to the Grunt capabilities discussed in the book.

To run the project with Gulp, use the following command.

```
$ gulp run
```

To prepare the project for distribution to a server, use the following command.

```
$ gulp dist
```

⁷⁸<http://gulpjs.com/>