PART IV

Rapid Application Development

CHAPTER 9

Persistence Layers with Spring Roo

Spring Roo is a rapid application development tool for Java developers. With Roo, you can easily build full Java applications in minutes.

We won't be covering all aspects of Roo development in this chapter. We will focus on the new repository support for JPA and MongoDB that uses Spring Data to provide this support. If you want to read more about Roo, go to the Spring Roo project home page, where you can find links to the reference manual. While on the project home page, look for a link to download a free O'Reilly ebook by Josh Long and Steve Mayzak called *Getting Started with Roo* [LongMay11]. This ebook covers an older 1.1 version of Roo that does not support the repository layer, but it is a good introduction to Roo in general. The most up-to-date guide for using Spring Roo is *Spring Roo in Action* by Ken Rimple and Srini Penchikala [RimPen12].

A Brief Introduction to Roo

Roo works its magic using code generation combined with AspectJ for injecting behavior into your domain and web classes. When you work on a Roo project, the project files are monitored by Roo and additional artifacts are generated. You still have your regular Java classes that you can edit, but there are additional features provided for free. When you create a class with Roo and annotate that class with one or more annotations that provide additional capabilities, Roo will generate a corresponding AspectJ file that contains one or more AspectJ inter type declarations (ITD). There is, for instance, an `@RooJavaBean` annotation that triggers the generation of an AspectJ aspect declaration that provides ITDs that introduce getters and setters in your Java class. There's no need to code that yourself. Let's see a quick example of how that would look. Our simple bean class in shown in Example 9-1.

Example 9-1. A simple Java bean class: Address.java

```
@RooJavaBean
public class Address {
    private String street;

    private String city;

    private String country;
}
```

As you can see, we don't code the getters and setters. They will be introduced by the backing AspectJ aspect file since we used the `@RooJavaBean` annotation. The generated AspectJ file looks like Example 9-2.

Example 9-2. The generated AspectJ aspect definition: Address_Roo_JavaBean.aj

```
// WARNING: DO NOT EDIT THIS FILE. THIS FILE IS MANAGED BY SPRING ROO.
// You may push code into the target .java compilation unit if you wish to edit any member(s).

package com.oreilly.springdata.roo.domain;

import com.oreilly.springdata.roo.domain.Address;

privileged aspect Address_Roo_JavaBean {

    public String Address.getStreet() {
        return this.street;
    }

    public void Address.setStreet(String street) {
        this.street = street;
    }

    public String Address.getCity() {
        return this.city;
    }

    public void Address.setCity(String city) {
        this.city = city;
    }

    public String Address.getCountry() {
        return this.country;
    }

    public void Address.setCountry(String country) {
        this.country = country;
    }

}
```

You can see that this is defined as a privileged aspect, which means that it will have access to any private variables declared in the target class. The way you would define ITDs is by preceding any method names with the target class name, separated by a dot. So `public String Address.getStreet()` will introduce a new method in the `Address` class with a `public String getStreet()` signature.

As you can see, Roo follows a specific naming pattern that makes it easier to identify what files it has generated. To work with Roo, you can either use a command-line shell or edit your source files directly. Roo will synchronize all changes and maintain the source and generated files as necessary.

When you ask Roo to create a project for you, it generates a *pom.xml* file that is ready for you to use when you build the project with Maven. In this *pom.xml* file, there is a Maven compile as well as an AspectJ plug-in defined. This means that all the AspectJ aspects are woven at compile time. In fact, nothing from Roo remains in the Java class files that your build generates. There is no runtime jar dependency. Also, the Roo annotations are source-level retention only, so they will not be part of your class files. You can, in fact, easily get rid of Roo if you so choose. You have the option of pushing all of the code defined in the AspectJ files into the appropriate source files and removing any of these AspectJ files. This is called *push-in refactoring*, and it will leave you with a pure Java solution, just as if you had written everything from scratch yourself. Your application still retains all of the functionality.

Roo's Persistence Layers

Spring Roo started out supporting JPA as the only persistence option. It also was opinionated in terms of the data access layer. Roo prescribed an active record data access style where each entity provides its finder, save, and delete methods.

Starting with Roo version 1.2, we have additional options for the persistence layer (see Figure 9-1). Roo now allows you to choose between the default active record style and a repository-based persistence layer. If you choose the repository approach, you have a choice between JPA and MongoDB as the persistence providers. The actual repository support that Roo uses is the one provided by Spring Data, which we have already seen in Chapter 2.

In addition to an optional repository layer, Roo now also lets you define a service layer on top of either the active record style or repository style persistence layer.

Quick Start

You can use Roo either as a command-line tool or within an IDE, like the free Spring Tool Suite, that has built-in Roo support. Another IDE that has support for Roo is IntelliJ IDEA, but we won't be covering the support here.

Using Roo from the Command Line

First, you need to download the latest Spring Roo distribution from the download page. Once you have the file downloaded, unzip it somewhere on your system. In the *bin* directory, there is a *roo.sh* shell script for Unix-style systems as well as a *roo.bat*

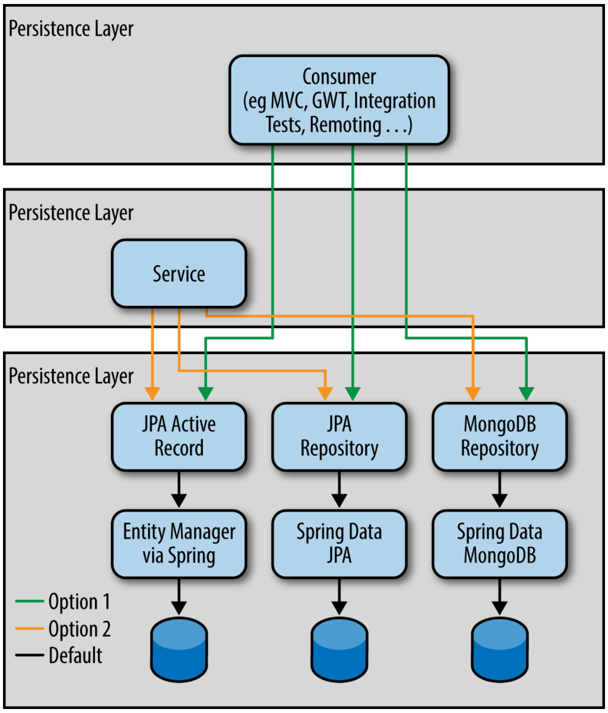


Figure 9-1. Spring Roo 1.2 layer architecture

batch file for Windows. When you want to create a Roo project, simply create a project directory and start Roo using the shell script or the batch file. If you add the *bin* directory to your path, you can just use the command name to start Roo; otherwise, you will have to provide the fully qualified path.

Once Roo starts up, you are greeted with the following screen (we entered `hint` at the prompt to get the additional information):

```
    ____  ____  ____
   / __ \/ __ \/ __ \
  / /_/ / / / / / / /
 / _, _/ /_/ / /_/ /
/_/ |_|\____/\____/    1.2.2.RELEASE [rev 7d75659]


Welcome to Spring Roo. For assistance press TAB or type "hint" then hit ENTER.
roo> hint
Welcome to Roo! We hope you enjoy your stay!

Before you can use many features of Roo, you need to start a new project.

To do this, type 'project' (without the quotes) and then hit TAB.
```

```
Enter a --topLevelPackage like 'com.mycompany.projectname' (no quotes).
When you've finished completing your --topLevelPackage, press ENTER.
Your new project will then be created in the current working directory.

Note that Roo frequently allows the use of TAB, so press TAB regularly.
Once your project is created, type 'hint' and ENTER for the next suggestion.
You're also welcome to visit http://forum.springframework.org for Roo help.
roo>
```

We are now ready to create a project and start developing our application. At any time, you can enter **`hint`**, and Roo will respond with some instruction on what to do next based on the current state of your application development. To cut down on typing, Roo will attempt to complete the commands you enter whenever you hit the Tab key.

Using Roo with Spring Tool Suite

The Spring Tool Suite comes with built-in Roo support, and it also comes bundled with Maven and the Developer Edition of *VMware vFabric tc Server*. This means that you have everything you need to develop applications with Roo. Just create your first Roo application using the menu option File→New→Spring Roo Project. You can see this in action in Figure 9-2.

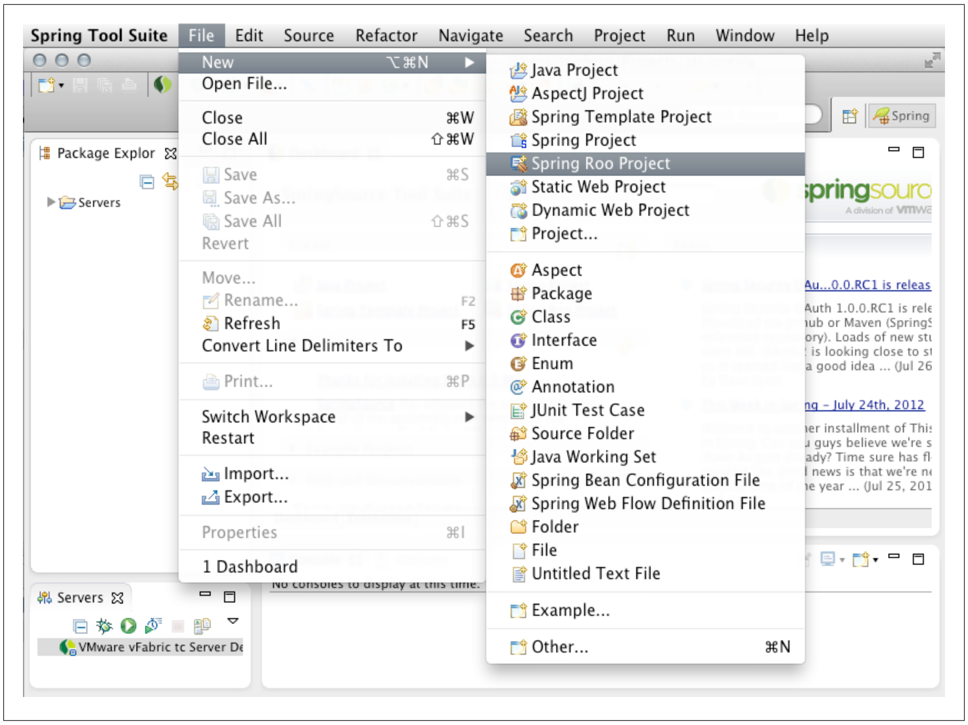
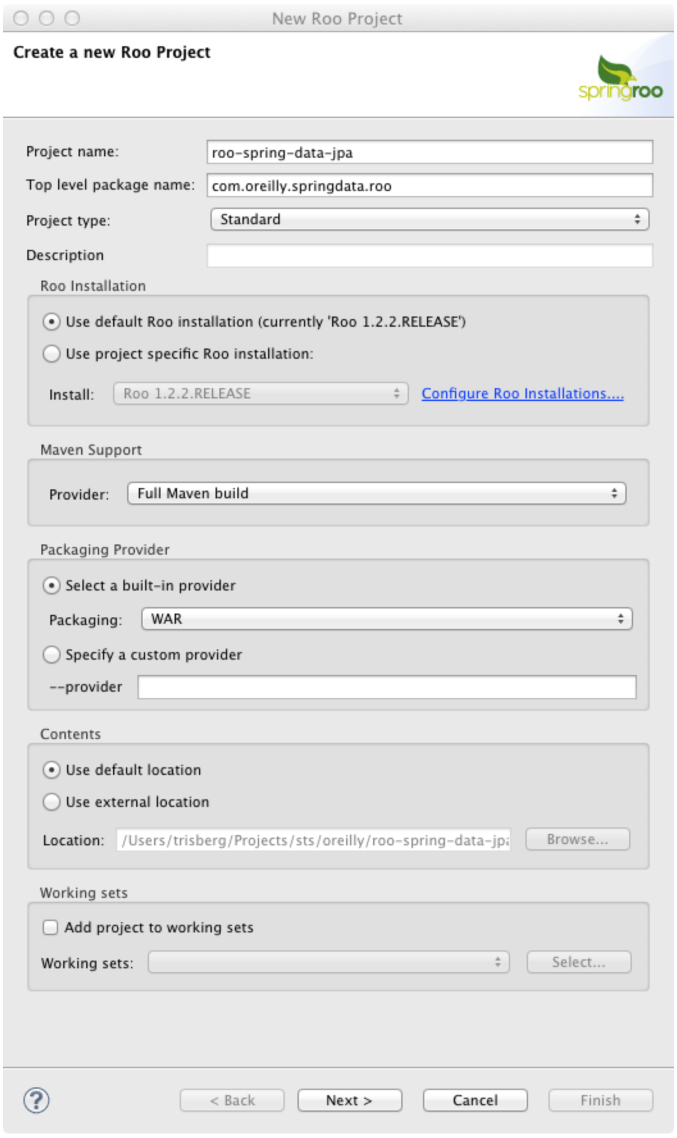


Figure 9-2. Creating a Spring Roo project—menu option

This opens a “Create a new Roo Project” dialog screen, as shown in Figure 9-3.



Project name: roo-spring-data-jpa
Top level package name: com.oreilly.springdata.roo
Project type: Standard
Description

Roo Installation
(●) Use default Roo installation (currently 'Roo 1.2.2.RELEASE')
( ) Use project specific Roo installation:
Install: Roo 1.2.2.RELEASE   Configure Roo Installations....

Maven Support
Provider: Full Maven build

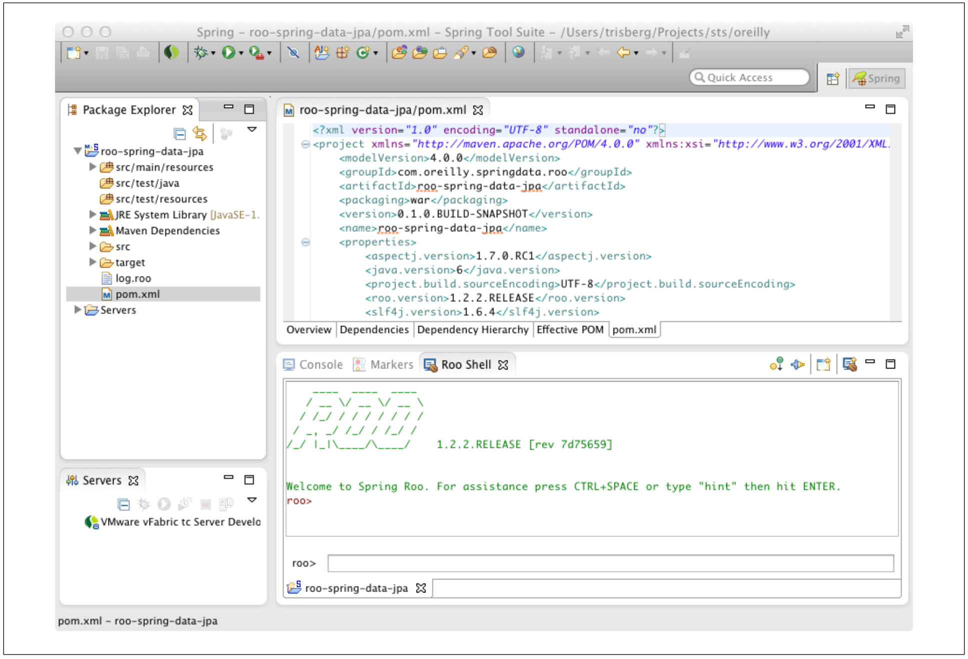
Packaging Provider
(●) Select a built-in provider
Packaging: WAR
( ) Specify a custom provider
--provider

Contents
(●) Use default location
( ) Use external location
Location: /Users/trisberg/Projects/sts/oreilly/roo-spring-data-jp

Working sets
[ ] Add project to working sets
Working sets:

Figure 9-3. Creating a Spring Roo project—new project dialog

Just fill in the “Project name” and “Top level package name,” and then select WAR as the packaging. Click Next, and then click Finish on the next screen. The project should now be created, and you should also see the Roo shell window, as shown in Figure 9-4.



```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XML.
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.oreilly.springdata.roo</groupId>
    <artifactId>roo-spring-data-jpa</artifactId>
    <packaging>war</packaging>
    <version>0.1.0.BUILD-SNAPSHOT</version>
    <name>roo-spring-data-jpa</name>
    <properties>
        <aspectj.version>1.7.0.RC1</aspectj.version>
        <java.version>6</java.version>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <roo.version>1.2.2.RELEASE</roo.version>
        <slf4j.version>1.6.4</slf4j.version>
```

```
    ____  ____  ____
   / __ \/ __ \/ __ \
  / /_/ / / / / / / /
 / _, _/ /_/ / /_/ /
/_/ |_|\____/\____/    1.2.2.RELEASE [rev 7d75659]


Welcome to Spring Roo. For assistance press CTRL+SPACE or type "hint" then hit ENTER.
roo>
```

```
roo>
```

Figure 9-4. Creating a Spring Roo project—new project with Roo Shell

A Spring Roo JPA Repository Example

We are now ready to build the first Roo project. We will start with a customer service application based on the same domain model that we have seen in earlier chapters. We will create a `Customer` class and an associated `Address` class, link them together, and create repositories and really basic data entry screens for them. Since Roo's repository support supports both JPA and MongoDB, using the Spring Data repository support, we will create one of each kind of application. As you will see, they are almost identical, but there are a couple of differences that we will highlight. So, let's get started. We'll begin with the JPA application.

Creating the Project

If you are using Spring Tool Suite, then just follow the aforementioned instructions to create a new Spring Roo project. On the "Create a new Roo Project" dialog screen, provide the following settings:

- Project name: `roo-spring-data-jpa`
- Top level package name: `com.oreilly.springdata.roo`
- Packaging: `WAR`

If you are using the command-line Roo shell, you need to create a *roo-spring-data-jpa* directory; once you change to this new directory, you can start the Roo shell as just explained. At the `roo>` prompt, enter the following command:

```
project --topLevelPackage com.oreilly.springdata.roo ↪
  --projectName roo-spring-data-jpa --java 6 --packaging WAR
```

You now have created a new project, and we are ready to start developing the application. From here on, the actions will be the same whether you are using the Roo shell from the command line or inside the Spring Tool Suite.

Setting Up JPA Persistence

Setting up the JPA persistence configuration consists of selecting a JPA provider and a database. We will use Hibernate together with HSQLDB for this example. At the `roo>` prompt, enter the following:

```
jpa setup --provider HIBERNATE --database HYPERSONIC_PERSISTENT
```

Remember that when entering these commands, you can always press the Tab key to get completion and suggestions for available options. If you are using the Spring Tool Suite, press Ctrl+Space instead.

Creating the Entities

Let's create our entities, starting with the `Address` class:

```
entity jpa --class ~.domain.Address --activeRecord false
field string --fieldName street --notNull
field string --fieldName city --notNull
field string --fieldName country --notNull
```

That wasn't too hard. Note that we specified `--activeRecord false`, which means that we will have to provide the CRUD functionality using a repository. The resulting `Address` class looks like this:

```
package com.oreilly.springdata.roo.domain;

import javax.validation.constraints.NotNull;
import org.springframework.roo.addon.javabean.RooJavaBean;
import org.springframework.roo.addon.jpa.entity.RooJpaEntity;
import org.springframework.roo.addon.tostring.RooToString;

@RooJavaBean
@RooToString
@RooJpaEntity
public class Address {

    @NotNull
    private String street;
    @NotNull
    private String city;

    @NotNull
    private String country;
}
```

We see the private fields we declared and three Roo annotations: `@RooJavaBean`, `@RooToString`, and `@RooJpaEntity`. These annotations have corresponding AspectJ aspect declarations that you can find in the same directory as the Java class.

Let's create the `EmailAddress` and `Customer` classes next. The `EmailAddress` is an embeddable class with a single `value` field. We need to ask Roo to ignore the fact that *value* is a reserved word for some SQL databases. We also provide a column name of `email` since that will be more descriptive for anyone inspecting the database table. Using this `embeddable` in a field declarations, we specify it as an `embedded` field.

```
embeddable --class ~.domain.EmailAddress
field string --fieldName value --notNull --column email --permitReservedWords
entity jpa --class ~.domain.Customer --activeRecord false
field string --fieldName firstName --notNull
field string --fieldName lastName --notNull
field embedded --fieldName emailAddress --type ~.domain.EmailAddress
field set --fieldName addresses --type ~.domain.Address
```

The last command creates a many-to-many relationship to the address table, allowing us to provide a number of addresses for each customer. This is the resulting `Customer` class:

```
package com.oreilly.springdata.roo.domain;

import java.util.HashSet;
import java.util.Set;
import javax.persistence.CascadeType;
import javax.persistence.Embedded;
import javax.persistence.ManyToMany;
import javax.validation.constraints.NotNull;
import org.springframework.roo.addon.javabean.RooJavaBean;
import org.springframework.roo.addon.jpa.entity.RooJpaEntity;
import org.springframework.roo.addon.tostring.RooToString;

@RooJavaBean
@RooToString
@RooJpaEntity
public class Customer {

    @NotNull
    private String firstName;

    @NotNull
    private String lastName;

    @Embedded
    private EmailAddress emailAddress;
    @ManyToMany(cascade = CascadeType.ALL)
    private Set<Address> addresses = new HashSet<Address>();
}
```

For completeness, we'll show the `EmailAddress` class as well:

```
package com.oreilly.springdata.roo.domain;

import javax.persistence.Column;
import javax.persistence.Embeddable;
import javax.validation.constraints.NotNull;
import org.springframework.roo.addon.javabean.RooJavaBean;
import org.springframework.roo.addon.tostring.RooToString;

@RooJavaBean
@RooToString
@Embeddable
public class EmailAddress {

    @NotNull
    @Column(name = "email")
    private String value;
}
```

The most interesting part of this class is that it is `@Embeddable` and that we have defined the value property to be stored in a database column named `email`.

Defining the Repositories

With the entities in place, we can now create the JPA repositories. We create one repository for each entity.

```
repository jpa --interface ~.repository.CustomerRepository --entity ~.domain.Customer
repository jpa --interface ~.repository.AddressRepository --entity ~.domain.Address
```

At this point we could also create a service layer, but since this is such a simple application, we'll skip this step.

Creating the Web Layer

Now we need some really simple web pages so we can enter and modify our customer and address data. We'll just stick with the screens generated by Roo.

```
web mvc setup
web mvc scaffold --class ~.web.CustomerController --backingType ~.domain.Customer
web mvc scaffold --class ~.web.AddressController --backingType ~.domain.Address
```

There is one thing we have to do. Roo doesn't know how to map the `EmailAddress` class between the `String` representation used for web pages and the `EmailAddress` type used for persistence. We need to add converters to the `ApplicationConversionServiceFactoryBean` that Roo generated; Example 9-3 shows how.

Example 9-3. The generated ApplicationConversionServiceFactoryBean.java with converters added

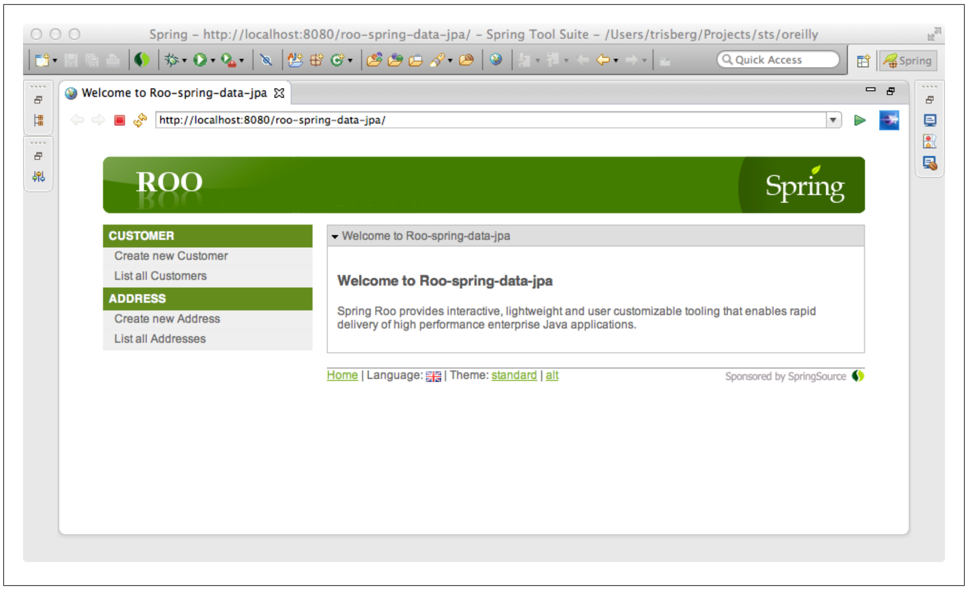
```
package com.oreilly.springdata.roo.web;

import org.springframework.core.convert.converter.Converter;
import org.springframework.format.FormatterRegistry;
import org.springframework.format.support.FormattingConversionServiceFactoryBean;
import org.springframework.roo.addon.web.mvc.controller.converter.RooConversionService;

import com.oreilly.springdata.roo.domain.EmailAddress;

/**
 * A central place to register application converters and formatters.
 */
@RooConversionService
public class ApplicationConversionServiceFactoryBean
        extends FormattingConversionServiceFactoryBean {

  @Override
  protected void installFormatters(FormatterRegistry registry) {
    super.installFormatters(registry);
    // Register application converters and formatters
    registry.addConverter(getStringToEmailAddressConverter());
    registry.addConverter(getEmailAddressConverterToString());
  }

  public Converter<String, EmailAddress> getStringToEmailAddressConverter() {
    return new Converter<String, EmailAddress>() {
      @Override
      public EmailAddress convert(String source) {
        EmailAddress emailAddress = new EmailAddress();
        emailAddress.setAddress(source);
        return emailAddress;
      }
    };
  }

  public Converter<EmailAddress, String> getEmailAddressConverterToString() {
    return new Converter<EmailAddress, String>() {
      @Override
      public String convert(EmailAddress source) {
        return source.getAddress();
      }
    };
  }
}
```

Running the Example

Now we are ready to build and deploy this example. For Spring Tool Suite, just drag the application to the tc server instance and start the server. If you use the command line, simply exit the Roo shell and from the command line run the following Maven commands:

```
mvn clean package
mvn tomcat:run
```

You should now be able to open a browser and navigate to *http://localhost:8080/roo-spring-data-jpa/* and see the screen shown in Figure 9-5.


Spring – http://localhost:8080/roo-spring-data-jpa/ – Spring Tool Suite – /Users/trisberg/Projects/sts/oreilly
Quick Access
Spring
Welcome to Roo-spring-data-jpa
http://localhost:8080/roo-spring-data-jpa/
ROO
Spring
CUSTOMER
Create new Customer
List all Customers
ADDRESS
Create new Address
List all Addresses
Welcome to Roo-spring-data-jpa
Home | Language: | Theme: standard | alt
Sponsored by SpringSource

Welcome to Roo-spring-data-jpa

Spring Roo provides interactive, lightweight and user customizable tooling that enables rapid delivery of high performance enterprise Java applications.

Figure 9-5. The JPA application

Our application is now complete, and we can add some addresses and then a customer or two.

> If you get tired of losing your data every time you restart your app server, you can change the schema creation properties in *src/main/resources/META-INF/persistence.xml*. Change `<property name="hibernate.hbm2ddl.auto" value="create" />` to have a value of `"update"`.

A Spring Roo MongoDB Repository Example

Since Spring Data includes support for MongoDB repositories, we can use MongoDB as a persistence option when using Roo. We just won't have the option of using the active record style for the persistence layer; we can only use the repositories. Other than this difference, the process is very much the same as for a JPA solution.

Creating the Project

If you are using Spring Tool Suite, then just follow the aforementioned instructions to create a new Spring Roo project. On the "Create a new Roo Project" dialog screen, provide the following settings:

- Project name: `roo-spring-data-mongo`
- Top level package name: `com.oreilly.springdata.roo`
- Packaging: `WAR`

When using the command-line Roo shell, create a *roo-spring-data-mongo* directory. Change to this new directory and then start the Roo Shell as previously explained. At the `roo>` prompt, enter the following command:

```
project --topLevelPackage com.oreilly.springdata.roo ↪
  --projectName roo-spring-data-mongo --java 6 --packaging WAR
```

Setting Up MongoDB Persistence

Setting up the persistence configuration for MongoDB is simple. We can just accept the defaults. If you wish, you can provide a host, port, username, and password, but for a default local MongoDB installation the defaults work well. So just enter the following:

```
mongo setup
```

Creating the Entities

When creating the entities, we don't have the option of using the active record style, so there is no need to provide an `--activeRecord` parameter to opt out of it. Repositories are the default, and the only option for the persistence layer with MongoDB. Again, we start with the `Address` class:

```
entity mongo --class ~.domain.Address
field string --fieldName street --notNull
field string --fieldName city --notNull
field string --fieldName country --notNull
```

That looks very similar to the JPA example. When we move on to the `Customer` class, the first thing you'll notice that is different is that with MongoDB you don't use an `embeddable` class. That is available only for JPA. With MongoDB, you just create a plain `class` and specify `--rooAnnotations true` to enable the `@RooJavaBean` support. To use this class, you specify the field as `other`. Other than these minor differences, the entity declaration is very similar to the JPA example:

```
class --class ~.domain.EmailAddress --rooAnnotations true
field string --fieldName value --notNull --permitReservedWords
entity mongo --class ~.domain.Customer
field string --fieldName firstName --notNull
field string --fieldName lastName --notNull
field other --fieldName emailAddress --type ~.domain.EmailAddress
field set --fieldName addresses --type ~.domain.Address
```

Defining the Repositories

We declare the MongoDB repositories the same way as the JPA repositories except for the `mongo` keyword:

```
repository mongo --interface ~.repository.CustomerRepository ↪
  --entity ~.domain.Customer
repository mongo --interface ~.repository.AddressRepository --entity ~.domain.Address
```

Creating the Web Layer

The web layer is exactly the same as for the JPA example:

```
web mvc setup
web mvc scaffold --class ~.web.CustomerController --backingType ~.domain.Customer
web mvc scaffold --class ~.web.AddressController --backingType ~.domain.Address
```

Don't forget to add the converters to the `ApplicationConversionServiceFactoryBean` like we did for JPA in Example 9-3.

Running the Example

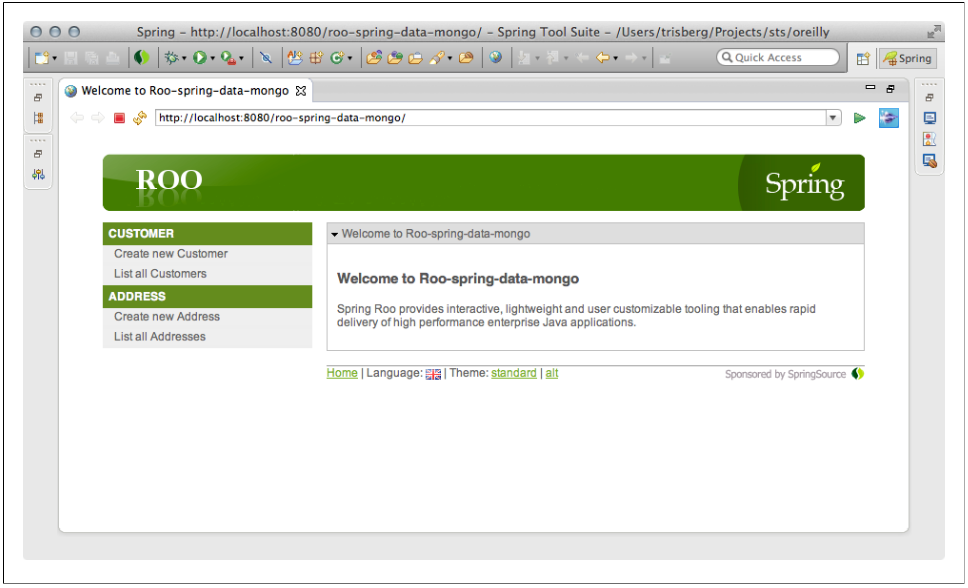
Now we are ready to build and deploy this example. This is again exactly the same as the JPA example, except that we need to have MongoDB running on our system. See Chapter 6 for instructions on how to install and run MongoDB.

For Spring Tool Suite, just drag the application to the tc server instance and start the server. If you use the command line, simply exit the Roo shell and from the command line run the following Maven commands:

```
mvn clean package
mvn tomcat:run
```

You should now be able to open a browser and navigate to *http://localhost:8080/roo-spring-data-mongo/* and see the screen in Figure 9-6.

Our second example application is now complete, and we can add some addresses and then a customer or two.


Spring – http://localhost:8080/roo-spring-data-mongo/ – Spring Tool Suite – /Users/trisberg/Projects/sts/oreilly
Quick Access
Spring
Welcome to Roo-spring-data-mongo
http://localhost:8080/roo-spring-data-mongo/
ROO
Spring
CUSTOMER
Create new Customer
List all Customers
ADDRESS
Create new Address
List all Addresses
Welcome to Roo-spring-data-mongo
Home | Language: | Theme: standard | alt
Sponsored by SpringSource

Welcome to Roo-spring-data-mongo

Spring Roo provides interactive, lightweight and user customizable tooling that enables rapid delivery of high performance enterprise Java applications.

Figure 9-6. The MongoDB application

CHAPTER 10

REST Repository Exporter

When you are working with the Spring Data repository abstraction (see Chapter 2 for details), the repository interface managing an entity becomes the central point of access for it. Using the Spring Data REST repository exporter project, you can now export (as the name suggests) these managed entities via a REST web service to easily interact with the data. The exporter mechanism will transparently expose a resource per repository, map CRUD operations onto HTTP methods for that resource, and provide a means to execute query methods exposed on the repository interface.

What Is REST?

Representational State Transfer (REST) is an architectural style initially described by Roy Fielding in his dissertation analyzing styles of network-based software architectures [Fielding00]. It is a generalization of the principles behind the HTTP protocol, from which it derives the following core concepts:

Resources
: Systems expose resources to other systems: an order, a customer, and so on.

Identifiers
: These resources can be addressed through an identifier. In the HTTP world, these identifiers are URIs.

Verbs
: Each resource can be accessed and manipulated though a well-defined set of verbs. These verbs have dedicated semantics and have to be used according to those. In HTTP the commonly used verbs are `GET`, `PUT`, `POST`, `DELETE`, `HEAD`, and `OPTIONS`, as well as the rather seldomly used (or even unused) ones, `TRACE` and `CONNECT`. Not every resource has to support all of the verbs just listed, but it is required that you don't set up special verbs per resource.

Representations
: A client never interacts with the resource directly but rather through representations of it. Representations are defined through media types, which clearly identify the structure of the representation. Commonly used media types include rather

general ones like `application/xml` and `application/json`, and more structured ones like `application/atom+xml`.

Hypermedia
The representations of a resource usually contain links to point to other resources, which allows the client to navigate the system based on the resource state and the links provided. This concept is described as Hypermedia as the Engine of Application State (HATEOAS).

Web services built based on these concepts have proven to be scalable, reliable, and evolvable. That's why REST web services are a ubiquitous means to integrate software systems. While Fielding's dissertation is a nice read, we recommend also looking at *REST in Practice*, by Jim Webber, Savas Parastatidis, and Ian Robinson. It provides a very broad, detailed, and real-world-example-driven introduction to the topic [WePaRo10].

We will have a guided tour through that functionality by walking through the *rest* module of the sample project. It is a Servlet 3.0–compliant, Spring-based web application. The most important dependency of the project is the *spring-data-rest-webmvc* library, which provides Spring MVC integration to export Spring Data JPA repositories to the Web. Currently, it works for JPA-backed repositories only, but support for other stores is on the roadmap already. The basic Spring infrastructure of the project is very similar to the one in Chapter 4, so if you haven't had a glance at that chapter already, please do so now to understand the basics.

The Sample Project

The easiest way to run the sample app is from the command line using the Maven Jetty plug-in. Jetty is a tiny servlet container capable of running Servlet 3.0 web applications. The Maven plug-in will allow you to bootstrap the app from the module folder using the command shown in Example 10-1.

Example 10-1. Running the sample application from the command line

```
$ mvn jetty:run -Dspring.profiles.active=with-data

[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Spring Data Book - REST exporter 1.0.0.BUILD-SNAPSHOT
[INFO] ------------------------------------------------------------------------
…
[INFO] <<< jetty-maven-plugin:8.1.5.v20120716:run (default-cli) @ spring-data-book-rest <<<
[INFO]
[INFO] --- jetty-maven-plugin:8.1.5.v20120716:run (default-cli) @ spring-data-book-rest ---
[INFO] Configuring Jetty for project: Spring Data Book - REST exporter
[INFO] webAppSourceDirectory …/spring-data-book/rest/src/main/webapp does not exist. \
       Defaulting to …/spring-data-book/rest/src/main/webapp
[INFO] Reload Mechanic: automatic
```

```
[INFO] Classes = …/spring-data-book/rest/target/classes
[INFO] Context path = /
[INFO] Tmp directory = …/spring-data-book/rest/target/tmp
[INFO] Web defaults = org/eclipse/jetty/webapp/webdefault.xml
[INFO] Web overrides =  none
[INFO] web.xml file = null
[INFO] Webapp directory = …/spring-data-book/rest/src/main/webapp
2012-07-31 17:58:01.709:INFO:oejs.Server:jetty-8.1.5.v20120716
2012-07-31 17:58:03.769:INFO:oejpw.PlusConfiguration:No Transaction manager found \
       - if your webapp requires one, please configure one.
2012-07-31 17:58:10.641:INFO:/:Spring WebApplicationInitializers detected on classpath: \
       [com.oreilly.springdata.rest.RestWebApplicationInitializer@34cbbc24]
…
2012-07-31 17:58:16,032  INFO est.webmvc.RepositoryRestExporterServlet: 444 - \
       FrameworkServlet 'dispatcher': initialization started
…
2012-07-31 17:58:17,159  INFO est.webmvc.RepositoryRestExporterServlet: 463 - \
       FrameworkServlet 'dispatcher': initialization completed in 1121 ms
2012-07-31 17:58:17.179:INFO:oejs.AbstractConnector:Started SelectChannelConnector@
    0.0.0.0:8080
[INFO] Started Jetty Server
```

The first thing to notice here is that we pipe a JVM argument, `spring.profiles.active`, into the execution. This populates the in-memory database with some sample products, customers, and orders so that we actually have some data to interact with. Maven now dumps some general activity information to the console. The next interesting line is the one at 17:58:10, which tells us that Jetty has discovered a `WebApplicationInitializer`—our `RestWebApplicationInitializer` in particular. A `WebApplicationInitializer` is the API equivalent to a *web.xml* file, introduced in the Servlet API 3.0. It allows us to get rid of the XML-based way to configure web application infrastructure components and instead use an API. Our implementation looks like Example 10-2.

Example 10-2. The RestWebApplicationInitializer

```java
public class RestWebApplicationInitializer implements WebApplicationInitializer {

  public void onStartup(ServletContext container) throws ServletException {

    // Create the 'root' Spring application context
    AnnotationConfigWebApplicationContext rootContext =
      new AnnotationConfigWebApplicationContext();
    rootContext.register(ApplicationConfig.class);

    // Manage the life cycle of the root application context
    container.addListener(new ContextLoaderListener(rootContext));

    // Register and map the dispatcher servlet
    DispatcherServlet servlet = new RepositoryRestExporterServlet();
    ServletRegistration.Dynamic dispatcher = container.addServlet("dispatcher", servlet);
    dispatcher.setLoadOnStartup(1);
    dispatcher.addMapping("/");
  }
}
```

First, we set up an AnnotationConfigWebApplicationContext and register the ApplicationConfig JavaConfig class to be used as a Spring configuration later on. We register that ApplicationContext wrapped inside ContextLoaderListener to the actual ServletContext. The context will invoke the listener later, which will cause the ApplicationContext to be bootstrapped in turn. The code so far is pretty much the equivalent of registering a ContextLoaderListener inside a *web.xml* file, pointing it to an XML config file, except that we don't have to deal with XML and String-based locations but rather type-safe references to configuration. The ApplicationContext configured will now bootstrap an embedded database, the JPA infrastructure including a transaction manager, and enable the repositories eventually. This process has already been covered in "Bootstrapping the Sample Code" on page 44.

Right after that, we declare a RepositoryRestExporterServlet, which actually cares about the tricky parts. It registers quite a bit of Spring MVC infrastructure components and inspects the root application context for Spring Data repository instances. It will expose HTTP resources for each of these repositories as long as they implement CrudRepository. This is currently a limitation, which will be removed in later versions of the module. We map the servlet to the servlet root so that the application will be available via http://localhost:8080 for now.

Interacting with the REST Exporter

Now that we have bootstrapped the application, let's see what we can actually do with it. We will use the command-line tool curl to interact with the system, as it provides a convenient way to trigger HTTP requests and displays responses in a way that we can show here in the book nicely. However, you can, of course, use any other client capable of triggering HTTP requests: command-line tools (like wget on Windows) or simply your web browser of choice. Note that the latter will only allow you to trigger GET requests through the URL bar. If you'd like to follow along with the more advanced requests (POST, PUT, DELETE), we recommend a browser plug-in like the Dev HTTP Client for Google Chrome. Similar tools are available for other browsers as well.

Let's trigger some requests to the application, as shown in Example 10-3. All we know right now is that we've deployed it to listen to *http://localhost:8080*, so let's see what this resource actually provides.

Example 10-3. Triggering the initial request using curl

```
$ curl -v http://localhost:8080

* About to connect() to localhost port 8080 (#0)
*   Trying ::1... connected
* Connected to localhost (::1) port 8080 (#0)
> GET / HTTP/1.1
> User-Agent: curl/7.21.4 (universal-apple-darwin11.0) libcurl/7.21.4 OpenSSL/0.9.8r
```

```
    zlib/1.2.5
> Host: localhost:8080
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Length: 242
< Content-Type: application/json
< Server: Jetty(8.1.5.v20120716)
<
{
  "links" : [ {
    "rel" : "product",
    "href" : "http://localhost:8080/product"
  }, {
    "rel" : "order",
    "href" : "http://localhost:8080/order"
  }, {
    "rel" : "customer",
    "href" : "http://localhost:8080/customer"
  } ]
}
* Connection #0 to host localhost left intact
* Closing connection #0
```

The first thing to notice here is that we triggered the `curl` command with the `-v` flag. This flag activates verbose output, listing all the request and response headers alongside the actual response data. We see that the server returns data of the content type `application/json` by default. The actual response body contains a set of links we can follow to explore the application. Each of the links provided is actually derived from a Spring Data repository available in the `ApplicationContext`. We have a `CustomerRepository`, a `ProductRepository`, and an `OrderRepository`, so the relation type (`rel`) attributes are `customer`, `product`, and `order` (the first part of the repository name beginning with a lowercase character). The resource URIs are derived using that default as well. To customize this behavior, you can annotate the repository interface with `@RestResource`, which allows you to explicitly define `path` (the part of the URI) as well as the `rel` (the relation type).

Links

The representation of a link is usually derived from the link element defined in the Atom RFC. It basically consists of two attributes: a relation type (`rel`) and a hypertext reference (`href`) . The former defines the actual semantics of the link (and thus has to be documented or standardized), whereas the latter is actually opaque to the client. A client will usually inspect a link's response body for relation types and follow the links with relation types it is interested in. So basically the client knows it will find all orders behind links with a `rel` of `order`. The actual URI is not relevant. This structure results in decoupling of the client and the server, as the latter tells the client where to go. This is especially useful in case a URI changes or the server actually wants to point the client to a different machine to load-balance requests.

Let's move on to inspecting the products available in the system. We know that the products are exposed via the relation type product; thus, we follow the link with that `rel`.

Accessing Products

Example 10-4 demonstrates how to access all products available in the system.

Example 10-4. Accessing products

```
$ curl http://localhost:8080/product

{ "content" : [ {
    "price" : 499.00,
    "description" : "Apple tablet device",
    "name" : "iPad",
    "links" : [ {
      "rel" : "self",
      "href" : "http://localhost:8080/product/1"
    } ],
    "attributes" : {
      "connector" : "socket"
    }
  }, … , {
    "price" : 49.00,
    "description" : "Dock for iPhone/iPad",
    "name" : "Dock",
    "links" : [ {
      "rel" : "self",
      "href" : "http://localhost:8080/product/3"
    } ],
    "attributes" : {
      "connector" : "plug"
    }
  } ],
  "links" : [ {
    "rel" : "product.search",
    "href" : "http://localhost:8080/product/search"
  } ]
}
```

Triggering the request to access all products returns a JSON representation containing two major fields. The `content` field consists of a collection of all available products rendered directly into the response. The individual elements contain the serialized properties of the `Product` class as well as an artificial `links` container. This container carries a single link with a relation type of `self`. The `self` type usually acts as a kind of identifier, as it points to the resource itself. So we can access the iPad product directly by following the link with the relation type `self` inside its representation (see Example 10-5).

Example 10-5. Accessing a single product

```
$ curl http://localhost:8080/product/1

{ "price" : 499.00,
  "description" : "Apple tablet device",
  "name" : "iPad",
  "links" : [ {
    "rel" : "self",
    "href" : "http://localhost:8080/product/1"
  } ],
  "attributes" : {
    "connector" : "socket"
  }
}
```

To update a product, you simply issue a PUT request to the resource providing the new content, as shown in Example 10-6.

Example 10-6. Updating a product

```
$ curl -v -X PUT -H "Content-Type: application/json" \
       -d '{ "price" : 469.00, \
             "name" : "Apple iPad" }' \
       http://localhost:8080/spring-data-book-rest/product/1

* About to connect() to localhost port 8080 (#0)
*   Trying ::1...
* connected
* Connected to localhost (::1) port 8080 (#0)
> PUT /spring-data-book-rest/product/1 HTTP/1.1
> User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 OpenSSL/0.9.8r
  zlib/1.2.5
> Host: localhost:8080
> Accept: */*
> Content-Type: application/json
> Content-Length: 82
>
* upload completely sent off: 82 out of 82 bytes
< HTTP/1.1 204 No Content
< Server: Apache-Coyote/1.1
< Date: Fri, 31 Aug 2012 10:23:58 GMT
```

We set the HTTP method to `PUT` using the `-X` parameter and provide a `Content-Type` header to indicate we're sending JSON. We submit an updated `price` and `name` attribute provided through the `-d` parameter. The server returns a `204 No Content` to indicate that the request was successful. Triggering another `GET` request to the product's URI returns the updated content, as shown in Example 10-7.

Example 10-7. The updated product

```
$ curl http://localhost:8080/spring-data-book-rest/product/1

{ "links" : [ {
    "rel" : "self",
    "href" : "http://localhost:8080/spring-data-book-rest"
  } ],
  "price" : 469.00,
  "description" : "Apple tablet device",
  "name" : "Apple iPad",
  "attributes" : {
    "connector" : "socket"
  }
}
```

The JSON representation of the collection resource also contained a `links` attribute, which points us to a generic resource that will allow us to explore the query methods exposed by the repository. The convention is using the relation type of the collection resource (in our case, `product`) extended by `.search`. Let's follow this link and see what searches we can actually execute—see Example 10-8.

Example 10-8. Accessing available searches for products

```
$ curl http://localhost:8080/product/search

{ "links" : [ {
    "rel" : "product.findByDescriptionContaining",
    "href" : "http://localhost:8080/product/search/findByDescriptionContaining"
  }, {
    "rel" : "product.findByAttributeAndValue",
    "href" : "http://localhost:8080/product/search/findByAttributeAndValue"
  } ]
}
```

As you can see, the repository exporter exposes a resource for each of the query methods declared in the `ProductRepository` interface. The relation type pattern is again based on the relation type of the resource extended by the query method name, but we can customize it using `@RestResource` on the query method. Since the JVM unfortunately doesn't support deriving the parameter names from interface methods, we have to annotate the method parameters of our query methods with `@Param` and use named parameters in our manual query method definition for `findByAttributeAndValue(...)`. See Example 10-9.

Example 10-9. The PersonRepository interface

```
public interface ProductRepository extends CrudRepository<Product, Long> {

  Page<Product> findByDescriptionContaining(
      @Param("description") String description, Pageable pageable);

  @Query("select p from Product p where p.attributes[:attribute] = :value")
  List<Product> findByAttributeAndValue(
      @Param("attribute") String attribute, @Param("value") String value);
}
```

We can now trigger the second query method by following the `product.findByAttributeAndValue` link and invoking a `GET` request handing the matching parameters to the server. Let's search for products that have a connector plug, as shown in Example 10-10.

Example 10-10. Searching for products with a connector attribute of value plug

```
$ curl http://localhost:8080/product/search/findByAttributeAndValue?attribute=connector\
      /&value=plug

{ "results" : [ {
    "price" : 49.00,
    "description" : "Dock for iPhone/iPad",
    "name" : "Dock",
    "_links" : [ {
      "rel" : "self",
      "href" : "http://localhost:8080/product/3"
    } ],
    "attributes" : {
      "connector" : "plug"
    }
  } ],
  "links" : [ … ]
}
```

Accessing Customers

Now that we've seen how to navigate through the products available and how to execute the finder methods exposed by the repository interface, let's switch gears and have a look at the customers registered in the system. Our original request to *http://localhost:8080* exposed a `customer` link (see Example 10-3). Let's follow that link in Example 10-11 and see what customers we find.

Example 10-11. Accessing customers (1 of 2)

```
$ curl -v http://localhost:8080/customer

* About to connect() to localhost port 8080 (#0)
*   Trying ::1...
* connected
* Connected to localhost (::1) port 8080 (#0)
> GET /customer HTTP/1.1
> User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 OpenSSL/0.9.8r
    zlib/1.2.5
> Host: localhost:8080
> Accept: */*
>
< HTTP/1.1 500 Could not write JSON: No serializer found for class
  com.oreilly.springdata.rest.core.EmailAddress and no properties
  discovered to create BeanSerializer …
< Content-Type: text/html;charset=ISO-8859-1
< Cache-Control: must-revalidate,no-cache,no-store
< Content-Length: 16607
< Server: Jetty(8.1.5.v20120716)
```

Yikes, that doesn't look too good. We're getting a `500 Server Error` response, indicating that something went wrong with processing the request. Your terminal output is probably even more verbose, but the important lines are listed in Example 10-11 right underneath the HTTP status code. Jackson (the JSON marshalling technology used by Spring Data REST) seems to choke on serializing the `EmailAddress` value object. This is due to the fact that we don't expose any getters or setters, which Jackson uses to discover properties to be rendered into the response.

Actually, we don't even want to render the `EmailAddress` as an embedded object but rather as a plain `String` value. We can achieve this by customizing the rendering using the `@JsonSerialize` annotation provided by Jackson. We configure its using attribute to the predefined `ToStringSerializer.class`, which will simply render the object by calling the `toString()` method on the object.

All right, let's give it another try (Example 10-12).

Example 10-12. Accessing customers (2 of 2)

```
$ curl -v http://localhost:8080/customer

* About to connect() to localhost port 8080 (#0)
*   Trying ::1...
* connected
* Connected to localhost (::1) port 8080 (#0)
> GET /customer HTTP/1.1
> User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 OpenSSL/0.9.8r
    zlib/1.2.5
> Host: localhost:8080
> Accept: */*
>
< HTTP/1.1 500 Could not write JSON: Infinite recursion (StackOverflowError)
  (through reference chain: com.oreilly.springdata.rest.core.Address["copy"]
  ->com.oreilly.springdata.rest.core.Address["copy"]…
< Content-Type: text/html;charset=ISO-8859-1
< Cache-Control: must-revalidate,no-cache,no-store
< Content-Length: 622972
< Server: Jetty(8.1.5.v20120716)
```

Well, it's not that much better, but at least we seem to get one step further. This time the Jackson renderer complains about the `Address` class exposing a `copy` property, which in turn causes a recursion. The reason for this issue is that the `getCopy()` method of `Address` class follows the Java bean property semantics but is not a getter method in the classic sense. Rather, it returns a copy of the `Address` object to allow us to easily create a clone of an `Address` instance and assign it to an `Order` to shield against changes to the `Customer`'s `Address` leaking into an already existing `Order` (see Example 4-7). So we have two options here: we could either rename the method to not match the Java bean property convention or add an annotation to tell Jackson to simply ignore the property. We'll choose the latter for now, as we don't want to get into refactoring the client code. Thus, we use the `@JsonIgnore` annotation to exclude the `copy` property from rendering, as shown in Example 10-13.

Example 10-13. Excluding the copy property of the Address class from rendering

```
@Entity
public class Address extends AbstractEntity {

  private String street, city, country;

  …

  @JsonIgnore
  public Address getCopy() { … }
}
```

Having made this change, let's restart the server and invoke the request again (Example 10-14).

Example 10-14. Accessing customers after marshalling tweaks

```
$ curl http://localhost:8080/customer

{ "results" : [ {
    "links" : [ {
      "rel" : "self",
      "href" : "http://localhost:8080/customer/1"
    } ],
    "lastname" : "Matthews",
    "emailAddress" : "dave@dmband.com",
    "firstname" : "Dave",
    "addresses" : [ {
      "id" : 1,
      "street" : "27 Broadway",
      "city" : "New York",
      "country" : "United States"
    }, {
      "id" : 2,
      "street" : "27 Broadway",
      "city" : "New York",
      "country" : "United States"
    } ]
  }, … ],
  "links" : [ {
    "rel" : "customer.search",
    "href" : "http://localhost:8080/customer/search"
  } ],
  "page" {
    "number" : 1,
    "size" : 20,
    "totalPages" : 1,
    "totalElements" : 3
  }
}
```

As you can see, the entities can now be rendered correctly. We also find the expected links section to point us to the available query methods for customers. What's new, though, is that we have an additional `page` attribute set in the returned JSON. It contains the current page number (`number`), the requested page size (`size`, defaulted to 20 here), the total number of pages available (`totalPages`), and the overall total number of elements available (`totalElements`).

These attributes appear due to the fact that our `CustomerRepository` extends `PagingAndSortingRepository` and thus allows accessing all customers on a page-by-page basis. For more details on that, have a look at "Defining Repositories" on page 19. This allows us to restrict the number of customers to be returned for the collection resource by using page and limit parameters when triggering the request. As we have three customers present, let's request an artificial page size of one customer, as shown in Example 10-15.

Example 10-15. Accessing the first page of customers

```
$ curl http://localhost:8080/customer?limit=1

{ "content" : [ … ],
  "links" : [ {
    "rel" : "customer.next",
    "href" : "http://localhost:8080/customer?page=2&limit=1"
  }, {
    "rel" : "customer.search",
    "href" : "http://localhost:8080/customer/search"
  } ],
  "page" : {
    "number" : 1,
    "size" : 1,
    "totalPages" : 3,
    "totalElements" : 3
  }
}
```

Note how the metainformation provided alongside the single result changed. The `totalPages` field now reflects three pages being available due to our selecting a page size of one. Even better, the server indicates that we can navigate the customers to the next page by following the `customer.next` link. It already includes the request parameters needed to request the second page, so the client doesn't need to construct the URI manually. Let's follow that link and see how the metadata changes while navigating through the collection (Example 10-16).

Example 10-16. Accessing the second page of customers

```
$ curl http://localhost:8080/customer?page=2\&limit=1

{ "content" : [ … ],
  "links" : [ {
    "rel" : "customer.prev",
    "href" : "http://localhost:8080/customer?page=1&limit=1"
  }, {
    "rel" : "customer.next",
    "href" : "http://localhost:8080/customer?page=3&limit=1"
  }, {
    "rel" : "customer.search",
    "href" : "http://localhost:8080/customer/search"
  } ],
  "page" : {
    "number" : 2,
    "size" : 1,
    "totalPages" : 3,
    "totalElements" : 3
  }
}
```

Note that you might have to escape the `&` when pasting the URI into the console shown in Example 10-16. If you're working with a dedicated HTTP client, escaping is not necessary.

Besides the actual content returned, note how the `number` attribute reflects our move to page two. Beyond that, the server detects that there is now a previous page available and offers to navigate to it through the `customer.prev` link. Following the `customer.next` link a second time would result in the next representation not listing the `customer.next` link anymore, as we have reached the end of the available pages.

Accessing Orders

The final root link relation to explore is `order`. As the name suggests, it allows us to access the `Orders` available in the system. The repository interface backing the resource is `OrderRepository`. Let's access the resource and see what gets returned by the server (Example 10-17).

Example 10-17. Accessing orders

```
$ curl http://localhost:8080/order

{ "content" : [ {
    "billingAddress" : {
      "id" : 2,
      "street" : "27 Broadway",
      "city" : "New York",
      "country" : "United States"
    },
    "shippingAddress" : {
      "id" : 2,
      "street" : "27 Broadway",
      "city" : "New York",
      "country" : "United States"
    },
    "lineItems" : [ … ]
    "links" : [ {
      "rel" : "order.Order.customer",
      "href" : "http://localhost:8080/order/1/customer"
    }, {
      "rel" : "self",
      "href" : "http://localhost:8080/order/1"
    } ],
  } ],
  "links" : [ {
    "rel" : "order.search",
    "href" : "http://localhost:8080/order/search"
  } ],
  "page" : {
    "number" : 1,
    "size" : 20,
    "totalPages" : 1,
    "totalElements" : 1
  }
}
```

The response contains a lot of well-known patterns we already have discussed: a link to point to the exposed query methods of the `OrderRepository`, and the nested content field, which contains serialized `Order` objects, inlined `Address` objects, and `LineItems`. Also, we see the pagination metadata due to `OrderRepository` implementing `PagingAndSortingRepository`.

The new thing to notice here is that the `Customer` instance held in the `Order` object is not inline but instead pointed to by a link. This is because `Customers` are managed by a Spring Data repository. Thus, they are exposed as subordinate resources of the `Order` to allow for manipulating the assignment. Let's follow that link and access the `Customer` who triggered the `Order`, as shown in Example 10-18.

Example 10-18. Accessing the Customer who placed the Order

```
$ curl http://localhost:8080/order/1/customer

{ "links" : [ {
    "rel" : "order.Order.customer.Customer",
    "href" : "http://localhost:8080/order/1/customer"
  }, {
    "rel" : "self",
    "href" : "http://localhost:8080/customer/1"
  } ],
  "emailAddress" : "dave@dmband.com",
  "lastname" : "Matthews",
  "firstname" : "Dave",
  "addresses" : [ {
    "street" : "27 Broadway",
    "city" : "New York",
    "country" : "United States"
  }, {
    "street" : "27 Broadway",
    "city" : "New York",
    "country" : "United States"
  } ]
}
```

This call returns the detailed information of the linked `Customer` and provides two links. The one with the `order.Order.customer.Customer` relation type points to the `Customer` connection resource, whereas the `self` link points to the actual `Customer` resource. What's the difference here? The former represents the assignment of the `Customer` to the order. We can alter this assignment by issuing a `PUT` request to the URI, and we could unassign it by triggering a `DELETE` request to it. In our case, the `DELETE` call will result in a `405 Method not allowed`, as the JPA mapping requires a `Customer` to be mapped via the `optional = false` flag in the `@ManyToOne` annotation of the `customer` property. If the relationship is optional, a `DELETE` request will just work fine.

Assume we discovered that it's actually not Dave who placed the `Order` initially, but Carter. How do we update the assignment? First, the HTTP method of choice is `PUT`, as we already know the URI of the resource to manipulate. It wouldn't really make sense to put actual data to the server, since all we'd like to do is tell the server "this existing `Customer` is the issuer of the `Order`." Because the `Customer` is identified through its URI, we're going to `PUT` it to the server, setting the `Content-Type` request header to `text/uri-list` so that the server knows what we send. See Example 10-19.

Example 10-19. Changing the Customer who placed an Order

```
$ curl -v -X PUT -H "Content-Type: text/uri-list" \
       -d "http://localhost:8080/customer/2" http://localhost:8080/order/1/customer

* About to connect() to localhost port 8080 (#0)
*   Trying ::1...
* connected
* Connected to localhost (::1) port 8080 (#0)
> PUT /order/1/customer HTTP/1.1
> User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 OpenSSL/0.9.8r zlib/1.2.5
> Host: localhost:8080
> Accept: */*
> Content-Type: text/uri-list
> Content-Length: 32
>
* upload completely sent off: 32 out of 32 bytes
< HTTP/1.1 204 No Content
< Content-Length: 0
< Content-Type: application/octet-stream
< Server: Jetty(8.1.5.v20120716)
```

`text/uri-list` is a standardized media type to define the format of one or more URIs being transferred. Note that we get a `204 No Content` from the server, indicating that it has accepted the request and completed the reassignment.

PART V
Big Data

CHAPTER 11

Spring for Apache Hadoop

Apache Hadoop is an open source project that originated in Yahoo! as a central component in the development of a new web search engine. Hadoop's architecture is based on the architecture Google developed to implement its own closed source web search engine, as described in two research publications that you can find here and here. The Hadoop architecture consists of two major components: a distributed filesystem and a distributed data processing engine that run on a large cluster of commodity servers. The Hadoop Distributed File System (HDFS) is responsible for storing and replicating data reliably across the cluster. Hadoop MapReduce is responsible for providing the programming model and runtime that is optimized to execute the code close to where the data is stored. The colocation of code and data on the same physical node is one of the key techniques used to minimize the time required to process large amounts (up to petabytes) of data.

While Apache Hadoop originated out of a need to implement a web search engine, it is a general-purpose platform that can be used for a wide variety of large-scale data processing tasks. The combination of open source software, low cost of commodity servers, and the real-world benefits that result from analyzing large amounts of new unstructured data sources (e.g., tweets, logfiles, telemetry) has positioned Hadoop to be a de facto standard for enterprises looking to implement big data solutions.

In this chapter, we start by introducing the "Hello world" application of Hadoop, wordcount. The wordcount application is written using the Hadoop MapReduce API. It reads text files as input and creates an output file with the total count of each word it has read. We will first show how a Hadoop application is traditionally developed and executed using command-line tools and then show how this application can be developed as a standard Java application and configured using dependency injection. To copy the input text files into HDFS and the resulting output file out of HDFS, we use Spring for Apache Hadoop's HDFS scripting features.

Challenges Developing with Hadoop

There are several challenges you will face when developing Hadoop applications. The first challenge is the installation of a Hadoop cluster. While outside the scope of this book, creating and managing a Hadoop cluster takes a significant investment of time, as well as expertise. The good news is that many companies are actively working on this front, and offerings such as Amazon's Elastic Map Reduce let you get your feet wet using Hadoop without significant upfront costs. The second challenge is that, very often, developing a Hadoop application does not consist solely of writing a single MapReduce, Pig, or Hive job, but rather it requires you to develop a complete data processing pipeline. This data pipeline consists of the following steps:

1. Collecting the raw data from many remote machines or devices.
2. Loading data into HDFS, often a continuous process from diverse sources (e.g., application logs), and event streams.
3. Performing real-time analysis on the data as it moves through the system and is loaded into HDFS.
4. Data cleansing and transformation of the raw data in order to prepare it for analysis.
5. Selecting a framework and programming model to write data analysis jobs.
6. Coordinating the execution of many data analysis jobs (e.g., workflow). Each individual job represents a step to create the final analysis results.
7. Exporting final analysis results from HDFS into structured data stores, such as a relational database or NoSQL databases like MongoDB or Redis, for presentation or further analysis.

Spring for Apache Hadoop along with two other Spring projects, Spring Integration and Spring Batch, provide the basis for creating a complete data pipeline solution that has a consistent configuration and programming model. While this topic is covered in Chapter 13, in this chapter we must start from the basics: how to interact with HDFS and MapReduce, which in itself provides its own set of challenges.

Command-line tools are currently promoted in Hadoop documentation and training classes as the primary way to interact with HDFS and execute data analysis jobs. This feels like the logical equivalent of using SQL*Plus to interact with Oracle. Using command-line tools can lead you down a path where your application becomes a loosely organized collection of bash, Perl, Python, or Ruby scripts. Command-line tools also require you to create ad-hoc conventions to parameterize the application for different environments and to pass information from one processing step to another. There should be an easy way to interact with Hadoop programmatically, as you would with any other filesystem or data access technology.

Spring for Apache Hadoop aims to simplify creating Hadoop-based applications in Java. It builds upon the Spring Framework to provide structure when you are writing Hadoop applications. It uses the familiar Spring-based configuration model that lets you take advantage of the powerful configuration features of the Spring container, such as property placeholder replacement and portable data access exception hierarchies. This enables you to write Hadoop applications in the same style as you would write other Spring-based applications.

Hello World

The classic introduction to programming with Hadoop MapReduce is the wordcount example. This application counts the frequency of words in text files. While this sounds simple to do using Unix command-line utilities such as `sed`, `awk`, or `wc`, what compels us to use Hadoop for this is how well this problem can scale up to match Hadoop's distributed nature. Unix command-line utilities can scale to many megabytes or perhaps a few gigabytes of data. However, they are a single process and limited by the disk transfer rates of a single machine, which are on the order of 100 MB/s. Reading a 1 TB file would take about two and a half hours. Using Hadoop, you can scale up to hundreds of gigabytes, terabytes, or even petabytes of data by distributing the data across the HDFS cluster. A 1 TB dataset spread across 100 machines would reduce the read time to under two minutes. HDFS stores parts of a file across many nodes in the Hadoop cluster. The MapReduce code that represents the logic to perform on the data is sent to the nodes where the data resides, executing close to the data in order to increase the I/O bandwidth and reduce latency of the overall job. This stage is the "Map" stage in MapReduce. To join the partial results from each node together, a single node in the cluster is used to "Reduce" the partial results into a final set of data. In the case of the wordcount example, the word counts accumulated on individual machines are combined into the final list of word frequencies.

The fun part of running wordcount is selecting some sample text to use as input. While it was surely not the intention of the original authors, Project Gutenberg provides an easily accessible means of downloading large amounts of text in the form of public domain books. Project Gutenberg is an effort to digitize the full text of public domain books and has over 39,000 books currently available. You can browse the project website and download a few classic texts using *wget*. In Example 11-1, we are executing the command in the directory */tmp/gutenberg/download*.

Example 11-1. Using wget to download books for wordcount

```
wget -U firefox  http://www.gutenberg.org/ebooks/4363.txt.utf-8
```

Now we need to get this data into HDFS using a HDFS shell command.

Before running the shell command, we need an installation of Hadoop. A good guide to setting up your own Hadoop cluster on a single machine is described in Michael Noll's excellent online tutorial.

We invoke an HDFS shell command by calling the `hadoop` command located in the *bin* directory of the Hadoop distribution. The Hadoop command-line argument `dfs` lets you work with HDFS and in turn is followed by traditional file commands and arguments, such as `cat` or `chmod`. The command to copy files from the local filesystem into HDFS is `copyFromLocal`, as shown in Example 11-2.

Example 11-2. Copying local files into HDFS

```
hadoop dfs -copyFromLocal /tmp/gutenberg/download /user/gutenberg/input
```

To check if the files were stored in HDFS, use the `ls` command, as shown in Example 11-3.

Example 11-3. Browsing files in HDFS

```
hadoop dfs -ls /user/gutenberg/input
```

To run the wordcount application, we use the example jar file that ships as part of the Hadoop distribution. The arguments for the application is the name of the application to run—in this case, wordcount—followed by the HDFS input directory and output directory, as shown in Example 11-4.

Example 11-4. Running wordcount using the Hadoop command-line utility

```
hadoop jar hadoop-examples-1.0.1.jar wordcount /user/gutenberg/input
  /user/gutenberg/output
```

After issuing this command, Hadoop will churn for a while, and the results will be placed in the directory */user/gutenberg/output*. You can view the output in HDFS using the command in Example 11-5.

Example 11-5. View the output of wordcount in HDFS

```
hadoop dfs -cat /user/gutenberg/output/part-r-00000
```

Depending on how many input files you have, there may be more than one output file. By default, output filenames follow the scheme shown in Example 11-5 with the last set of numbers incrementing for each additional file that is output. To copy the results from HDFS onto the local filesystem, use the command in Example 11-6.

Example 11-6. View the output of wordcount in HDFS

```
hadoop dfs -getmerge /user/gutenberg/output /tmp/gutenberg/output/wordcount.txt
```

If there are multiple output files in HDFS, the `getmerge` option merges them all into a single file when copying the data out of HDFS to the local filesystem. Listing the file contents shows the words in alphabetical order followed by the number of times they appeared in the file. The superfluous-looking quotes are an artifact of the implementation of the MapReduce code that tokenized the words. Sample output of the wordcount application output is shown in Example 11-7.

Example 11-7. Partial listing of the wordcount output file

```
> cat /tmp/gutenberg/output/wordcount.txt
A 2
"AWAY 1
"Ah, 1
"And 2
"Another 1
…
"By 2
"Catholicism" 1
"Cease 1
"Cheers 1
…
```

In the next section, we will peek under the covers to see what the sample application that is shipped as part of the Hadoop distribution is doing to submit a job to Hadoop. This will help you understand what's needed to develop and run your own application.

Hello World Revealed

There are a few things going on behind the scenes that are important to know if you want to develop and run your own MapReduce applications and not just the ones that come out of the box. To get an understanding of how the example applications work, start off by looking in the *META-INF/manifest.mf* file in *hadoop-examples.jar*. The manifest lists `org.apache.hadoop.examples.ExampleDriver` as the main class for Java to run. `ExampleDriver` is responsible for associating the first command-line argument, `wordcount`, with the Java class `org.apache.hadoop.examples.Wordcount` and executing the main method of `Wordcount` using the helper class `ProgramDriver`. An abbreviated version of `ExampleDriver` is shown in Example 11-8.

Example 11-8. Main driver for the out-of-the-box wordcount application

```
public class ExampleDriver {

  public static void main(String... args){

    int exitCode = -1;
    ProgramDriver pgd = new ProgramDriver();

    try {
      pgd.addClass("wordcount", WordCount.class,
                   "A map/reduce program that counts the words in the input files.");
      pgd.addClass("randomwriter", RandomWriter.class,
                   "A map/reduce program that writes 10GB of random data per node.");

      // additional invocations of addClass excluded that associate keywords
      // with other classes

      exitCode = pgd.driver(args);
    } catch(Throwable e) {
      e.printStackTrace();
    }

    System.exit(exitCode);
  }
}
```

The WordCount class also has a main method that gets invoked not directly by the JVM when starting, but when the driver method of ProgramDriver is invoked. The WordCount class is shown in Example 11-9.

Example 11-9. The wordcount main method invoked by the ProgramDriver

```
public class WordCount {

  public static void main(String... args) throws Exception {

    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();

    if (otherArgs.length != 2) {
      System.err.println("Usage: wordcount <in> <out>");
      System.exit(2);
    }

    Job job = new Job(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
    FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

This gets at the heart of what you have to know in order to configure and execute your own MapReduce applications. The necessary steps are: create a new Hadoop Configuration object, create a Job object, set several job properties, and then run the job using the method waitforCompletion(…). The Mapper and Reducer classes form the core of the logic to write when creating your own application.

While they have rather generic names, TokenizerMapper and IntSumReducer are static inner classes of the WordCount class and are responsible for counting the words and summing the total counts. They're demonstrated in Example 11-10.

Example 11-10. The Mapper and Reducer for the out-of-the-box wordcount application

```
public class WordCount {

  public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable>{
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context)
      throws IOException, InterruptedException {

      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }

  public static class IntSumReducer extends Reducer<Text,IntWritable,Text,IntWritable> {

    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values, Context context)
      throws IOException, InterruptedException {

      int sum = 0;

      for (IntWritable val : values) {
        sum += val.get();
      }

      result.set(sum);
      context.write(key, result);
    }
  }

  // … main method as shown before
}
```

Since there are many out-of-the-box examples included in the Hadoop distribution, the `ProgramDriver` utility helps to specify which Hadoop job to run based off the first command-line argument. You can also run the `WordCount` application as a standard Java main application without using the `ProgramDriver` utility. A few minor modifications related to command-line argument handling are all that you need. The modified `WordCount` class is shown in Example 11-11.

Example 11-11. A standalone wordcount main application

```
public class WordCount {

  // ... TokenizerMapper shown before
  // ... IntSumReducer shown before

  public static void main(String[] args) throws Exception {

    Configuration conf = new Configuration();
    if (args.length != 2) {
      System.err.println("Usage: <in> <out>");
      System.exit(2);
    }

    Job job = new Job(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

The sample application for this section is located in the directory *./hadoop/wordcount*. A Maven build file for `WordCount` application is also provided; this lets you run the `WordCount` application as part of a regular Java application, independent of using the Hadoop command-line utility. Using Maven to build your application and run a standard Java main application is the first step toward treating the development and deployment of Hadoop applications as regular Java applications, versus one that requires a separate Hadoop command-line utility to execute. The Maven build uses the Appassembler plug-in to generate Unix and Windows start scripts and to collect all the required dependencies into a local *lib* directory that is used as the classpath by the generated scripts.

To rerun the previous `WordCount` example using the same output direction, we must first delete the existing files and directory, as Hadoop does not allow you to write to a preexisting directory. The command `rmr` in the HDFS shell achieves this goal, as you can see in Example 11-12.

Example 11-12. Removing a directory and its contents in HDFS

```
hadoop dfs -rmr /user/gutenberg/output
```

To build the application, run Appassembler's assemble target and then run the generated wordcount shell script (Example 11-13).

Example 11-13. Building, running, and viewing the output of the standalone wordcount application

```
$ cd hadoop/wordcount
$ mvn clean package appassembler:assemble
$ sh ./target/appassembler/bin/wordcount hdfs://localhost:9000/user/gutenberg/input
     hdfs://localhost:9000/user/gutenberg/output

INFO: Total input paths to process : 1
INFO: Running job: job_local_0001
…
```

```
INFO:     Map output records=65692

$ hadoop dfs -cat /user/gutenberg/output/part-r-00000
"A 2
"AWAY 1
"Ah, 1
"And 2
"Another 1
"Are 2
"BIG 1
…
```

One difference to using the Hadoop command line is that the directories in HDFS need to be prefixed with the URL scheme `hdfs://` along with the hostname and port of the namenode. This is required because the Hadoop command line sets environment variables recognized by the Hadoop `Configuration` class that prepend this information to the paths passed in as command-line arguments. Note that other URL schemes for Hadoop are available. The `webhdfs` scheme is very useful because it provides an HTTP-based protocol to communicate with HDFS that does not require the client (our application) and the HDFS server to use the exact same version (down to the minor point release) of the HDFS libraries.

Hello World Using Spring for Apache Hadoop

If you have been reading straight through this chapter, you may well be wondering, what does all this have to do with Spring? In this section, we start to show the features that Spring for Apache Hadoop provides to help you structure, configure, and run Hadoop applications. The first feature we will examine is how to use Spring to configure and run Hadoop jobs so that you can externalize application parameters in separate configuration files. This lets your application easily adapt to running in different environments—such as development, QA, and production—without requiring any code changes.

Using Spring to configure and run Hadoop jobs lets you take advantage of Spring's rich application configuration features, such as property placeholders, in the same way you use Spring to configure and run other applications. The additional effort to set up a Spring application context might not seem worth it for such a simple application as wordcount, but it is rare that you'll build such simple applications. Applications will typically involve chaining together several HDFS operations and MapReduce jobs (or equivalent Pig and Hive scripts). Also, as mentioned in "Challenges Developing with Hadoop" on page 176, there are many other development activities that you need to consider in creating a complete data pipeline solution. Using Spring for Apache Hadoop as a basis for developing Hadoop applications gives us a foundation to build upon and to reuse components as our application complexity grows.

Let's start by running the version of `WordCount` developed in the previous section inside of the Spring container. We use the XML namespace for Hadoop to declare the location of the namenode and the minimal amount of information required to define a `org.apache.hadoop.mapreduce.Job` instance. (See Example 11-14.)

Example 11-14. Declaring a Hadoop job using Spring's Hadoop namespace

```
<configuration>
  fs.default.name=hdfs://localhost:9000
</configuration>

<job id="wordcountJob"
     input-path="/user/gutenberg/input"
     output-path="/user/gutenberg/output"
     mapper="org.apache.hadoop.examples.WordCount.TokenizerMapper"
     reducer="org.apache.hadoop.examples.WordCount.IntSumReducer"/>

<job-runner id="runner" job="wordcountJob" run-at-startup="true"/>
```

This configuration will create a singleton instance of an `org.apache.hadoop.mapreduce.Job` managed by the Spring container. Some of the properties that were set on the job class programmatically in the previous example can be inferred from the class signature of the `Mapper` and `Reducer`. Spring can determine that `outputKeyClass` is of the type `org.apache.hadoop.io.Text` and that `outputValueClass` is of type `org.apache.hadoop.io.IntWritable`, so we do not need to set these properties explicitly. There are many other job properties you can set that are similar to the Hadoop command-line options (e.g., combiner, input format, output format, and general job key/value properties). Use XML schema autocompletion in Eclipse or another editor to see the various options available, and also refer to the Spring for Apache Hadoop reference documentation for more information. Right now, the namenode location, input, and output paths are hardcoded. We will extract them to an external property file shortly.

Similar to using the Hadoop command line to run a job, we don't need to specify the URL scheme and namenode host and port when specifying the input and output paths. The `<configuration/>` element defines the default URL scheme and namenode information. If you wanted to use the `webhdfs` protocol, then you simply need to set the value of the key `fs.default.name` to `webhdfs://localhost`. You can also specify values for other Hadoop configuration keys, such as `dfs.permissions`, `hadoop.job.ugi`, `mapred.job.tracker`, and `dfs.datanode.address`.

To launch a MapReduce job when a Spring `ApplicationContext` is created, use the utility class `JobRunner` to reference one or more managed `Job` objects and set the `run-at-startup` attribute to `true`. The main application class, which effectively takes the place of `org.apache.hadoop.examples.ExampleDriver`, is shown in Example 11-15. By default, the application looks in a well-known directory for the XML configuration file, but we can override this by providing a command-line argument that references another configuration location.

Example 11-15. Main driver to custom wordcount application managed by Spring

```
public class Main {

  private static final String[] CONFIGS = new String[] {
      "META-INF/spring/hadoop-context.xml" };

  public static void main(String[] args) {
    String[] res = (args != null && args.length > 0 ? args : CONFIGS);
    AbstractApplicationContext ctx = new ClassPathXmlApplicationContext(res);
    // shut down the context cleanly along with the VM
    ctx.registerShutdownHook();
  }
}
```

The sample code for this application is located in *./hadoop/wordcount-spring-basic*. You can build and run the application just like in the previous sections, as shown in Example 11-16. Be sure to remove the output files in HDFS that were created from running wordcount in previous sections.

Example 11-16. Building and running a basic Spring-based wordcount application

```
$ hadoop dfs -rmr /user/gutenberg/output
$ cd hadoop/wordcount-spring-basic
$ mvn clean package appassembler:assemble
$ sh ./target/appassembler/bin/wordcount
```

Now that the Hadoop job is a Spring-managed object, it can be injected into any other object managed by Spring. For example, if we want to have the wordcount job launched in a web application, we can inject it into a Spring MVC controller, as shown in Example 11-17.

Example 11-17. Dependency injection of a Hadoop job in WebMVC controller

```
@Controller
public class WordController {
  private final Job mapReduceJob;

  @Autowired
  public WordService(Job mapReduceJob) {
    Assert.notNull(mapReducejob);
    this.mapReduceJob = mapReduceJob;
  }

  @RequestMapping(value = "/runjob", method = RequestMethod.POST)
  public void runJob() {
    mapReduceJob.waitForCompletion(false);
  }
}
```

To start and externalize the configuration parameters of the application, we use Spring's `property-placeholder` functionality and move key parameters to a configuration file (Example 11-18).

Example 11-18. Declaring a parameterized Hadoop job using Spring's Hadoop namespace

```
<context:property-placeholder location="hadoop-default.properties"/>

<configuration>
  fs.default.name=${hd.fs}
</configuration>

<job id="wordcountJob"
     input-path="${wordcount.input.path}"
     output-path="${wordcount.output.path}"
     mapper="org.apache.hadoop.examples.WordCount.TokenizerMapper"
     reducer="org.apache.hadoop.examples.WordCount.IntSumReducer"/>

<job-runner id="runner" job="wordcountJob" run-at-startup="true"/>
```

The variable names `hd.fs`, `wordcount.input.path`, and `wordcount.output.path` are specified in the configuration file *hadoop-default.properties*, as shown in Example 11-19.

Example 11-19. The property file, hadoop-default.properties, that parameterizes the Hadoop application for the default development environment

```
hd.fs=hdfs://localhost:9000
wordcount.input.path=/user/gutenberg/input/
wordcount.output.path=/user/gutenberg/output/
```

This file is located in the *src/main/resources* directory so that it is made available on the classpath by the build script. We also can create another configuration file, named *hadoop-qa.properties*, which will define the location of the namenode as configured in the QA environment. To run the example on the same machine, we change only the name of the output path, as shown in Chapter 11. In a real QA environment, the location of the HDFS cluster, as well as the HDFS input and output paths, would be different.

Example 11-20. The property file, hadoop-qa.properties, that parameterizes the Hadoop application for the QA environment

```
hd.fs=hdfs://localhost:9000
wordcount.input.path=/data/words/input
wordcount.output.path=/data/words/qa/output
```

To take advantage of Spring's environment support, which enables easy switching between different sets of configuration files, we change the `property-placeholder` definition to use the variable `${ENV}` in the name of the property file to load. By default, Spring will resolve variable names by searching through JVM system properties and then environment variables. We specify a default value for the variable by using the syntax `${ENV:<Default Value>}`. In Example 11-21, if the shell environment variable, `ENV`, is not set, the value `default` will be used for `${ENV}` and the property file `hadoop-default.properties` will be loaded.

Example 11-21. Referencing different configuration files for different runtime envrionments

```
<context:property-placeholder location="hadoop-${ENV:default}.properties"/>
```

To run the application from the command line for the QA environment, run the commands in Example 11-22. Notice how the shell environment variable (ENV) is set to qa.

Example 11-22. Building and running the intermediate Spring-based wordcount application in the QA environment

```
$ hadoop dfs -copyFromLocal /tmp/gutenberg/download /user/gutenberg/input
$ hadoop dfs -rmr /user/gutenberg/qa/output
$ cd hadoop/wordcount-spring-intermediate
$ mvn clean package appassembler:assemble
$ export ENV=qa
$ sh ./target/appassembler/bin/wordcount
```

As we have seen over the course of these examples, when rerunning a Hadoop application we always need to write out the results to an empty directory; otherwise, the Hadoop job will fail. In the development cycle, it's tedious to remember to do this before launching the application each time, in particular when inside the IDE. One solution is to always direct the output to a new directory name based on a timestamp (e.g., `/user/gutenberg/output/2012/6/30/14/30`) instead of a static directory name. Let's see how we can use Spring's HDFS scripting features to help us with this common task.

Scripting HDFS on the JVM

When developing Hadoop applications, you will quite often need to interact with HDFS. A common way to get started with it is to use the HDFS command-line shell. For example, here's how to get a list of the files in a directory:

```
hadoop dfs -ls /user/gutenberg/input
```

While that is sufficient for getting started, when writing Hadoop applications you often need to perform more complex chains of filesystem operations. For example, you might need to test if a directory exists; if it does, delete it, and copy in some new files. As a Java developer, you might feel that adding this functionality into bash scripts is a step backward. Surely there is a programmatic way to do this, right? Good news: there is. It is the HDFS filesystem API. The bad news is that the Hadoop filesystem API is not very easy to use. It throws checked exceptions and requires us to construct `Path` instances for many of its method arguments, making the calling structure more verbose and awkward than needed. In addition, the Hadoop filesystem API does not provide many of the higher-level methods available from the command line, such as `test` and `chmod`.

Spring for Apache Hadoop provides an intermediate ground. It provides a simple wrapper for Hadoop's `FileSystem` class that accepts `Strings` instead of `Path` arguments. More importantly, it provides an `FsShell` class that mimics the functionality in the command- line shell but is meant to be used in a programmatic way. Instead of printing out information to the console, FsShell methods return objects or collections that you can inspect and use programmatically. The FsShell class was also designed to integrate with JVM-based scripting languages, allowing you to fall back to a scripting style interaction model but with the added power of using JRuby, Jython, or Groovy instead of bash. Example 11-23 uses the FsShell from inside a Groovy script.

Example 11-23. Declaring a Groovy script to execute

```
<configuration>
  fs.default.name=hdfs://localhost:9000
</configuration>

<script location="org/company/basic-script.groovy"/>
```

The `<script/>` element is used to create an instance of FsShell, which is implicitly passed into the Groovy script under the variable name fsh. The Groovy script is shown in Example 11-24.

Example 11-24. Using a Groovy script to work with HDFS

```
srcDir = "/tmp/gutenberg/download/"

// use the shell (made available under variable fsh)
dir = "/user/gutenberg/input"
if (!fsh.test(dir)) {
   fsh.mkdir(dir)
   fsh.copyFromLocal(srcDir, dir)
   fsh.chmod(700, dir)
}
```

Additional options control when the script gets executed and how it is evaluated. To avoid executing the script at startup, set `run-at-startup="false"` in the `script` tag's element. To reevaluate the script in case it was changed on the filesystem, set `evaluate="IF_MODIFIED"` in the `script` tag's element.

You can also parameterize the script that is executed and pass in variables that are resolved using Spring's property placeholder functionality, as shown in Example 11-25.

Example 11-25. Configuring a parameterized Groovy script to work with HDFS

```
<context:property-placeholder location="hadoop.properties"/>

<configuration>
  fs.default.name=${hd.fs}
</configuration>

<script id="setupScript" location="copy-files.groovy">
  <property name="localSourceFile" value="${localSourceFile}"/>
  <property name="inputDir" value="${inputDir}"/>
  <property name="outputDir" value="${outputDir}"/>
</script>
```

The property file *hadoop.properties* and *copy-files.groovy* are shown in Examples 11-26 and 11-27, respectively.

Example 11-26. Property file containing HDFS scripting variables

```
hd.fs=hdfs://localhost:9000
localSourceFile=data/apache.log
inputDir=/user/gutenberg/input/word/
outputDir=
  #{T(org.springframework.data.hadoop.util.PathUtils).format('/output/%1$tY/%1$tm/%1$td')}
```

Spring for Apache Hadoop also provides a `PathUtils` class that is useful for creating time-based directory paths. Calling the static `format` method will generate a time-based path, based on the current date, that uses `java.util.Formatter`'s convention for formatting time. You can use this class inside Java but also reference it inside configuration properties by using Spring's expression langauge, SpEL. The syntax of SpEL is similar to Java, and expressions are generally just one line of code that gets evaluated. Instead of using the syntax `${...}` to reference a variable, use the syntax `#{...}` to evaluate an expression. In SpEL, the special 'T' operator is used to specify an instance of a `java.lang.Class`, and we can invoke static methods on the 'T' operator.

Example 11-27. Parameterized Groovy script to work with HDFS

```
if (!fsh.test(hdfsInputDir)) {
   fsh.mkdir(hdfsInputDir);
   fsh.copyFromLocal(localSourceFile, hdfsInputDir);
   fsh.chmod(700, hdfsInputDir)
}
if (fsh.test(hdfsOutputDir)) {
   fsh.rmr(hdfsOutputDir)
}
```

Similar to how `FsShell` makes the command-line HDFS shell operations easily available in Java or JVM scripting languages, the class `org.springframework.data.hadoop.fs.DistCp` makes the Hadoop command-line `distcp` operations easily available. The `distcp` utility is used for copying large amounts of files within a single Hadoop cluster or between Hadoop clusters, and leverages Hadoop itself to execute the copy in a distributed manner. The `distcp` variable is implicitly exposed to the script as shown in Example 11-28, which demonstrates a script embedded inside of the `<hdp:script/>` element and not in a separate file.

Example 11-28. Using distcp inside a Groovy script

```
<configuration>
  fs.default.name=hdfs://localhost:9000
</configuration>

<script language="groovy">
  distcp.copy("${src}", "${dest}")
</script>
```

In this case, the Groovy script is embedded inside the XML instead of referencing an external file. It is also important to note that we can use Spring's property placeholder functionality to reference variables such as `${src} and ${dst}` inside the script, letting you parameterize the scripts.

Combining HDFS Scripting and Job Submission

A basic Hadoop application will have some HDFS operations and some job submissions. To sequence these tasks, you can use the `pre-action` and `post-action` attributes of the `JobRunner` so that HDFS scription operations occur before and after the job submission. This is illustrated in Example 11-29.

Example 11-29. Using dependencies between beans to control execution order

```
<context:property-placeholder location="hadoop.properties"/>

<configuration>
  fs.default.name=${hd.fs}
</configuration>

<job id="wordcountJob"
     input-path="${wordcount.input.path}"
     output-path="${wordcount.output.path}"
     mapper="org.apache.hadoop.examples.WordCount.TokenizerMapper"
     reducer="org.apache.hadoop.examples.WordCount.IntSumReducer"/>

<script id="setupScript" location="copy-files.groovy">
  <property name="localSourceFile" value="${localSourceFile}"/>
  <property name="inputDir" value="${wordcount.input.path}"/>
  <property name="outputDir" value="${wordcount.output.path}"/>
</script>

<job-runner id="runner" run-at-startup="true"
            pre-action="setupScript"
            job="wordcountJob"/>
```

The `pre-action` attribute references the `setupScript` bean, which in turn references the *copy-files.groovy* script that will reset the state of the system such that this application can be run and rerun without any interaction on the command line required. Build and run the application using the commands shown in Example 11-30.

Example 11-30. Building and running the intermediate Spring-based wordcount application

```
$ hadoop dfs -rmr /user/gutenberg/output
$ cd hadoop/wordcount-hdfs-copy
$ mvn clean package appassembler:assemble
$ sh ./target/appassembler/bin/wordcount
```

Example 11-31 shows the `JobRunner` configured to execute several HDFS scripts before and after the execution of multiple Hadoop jobs. The `JobRunner` also implements Java's `Callable` interface, which makes executing the `JobRunner` easy using Java's Executor framework. While this approach is convenient for executing simple workflows, it only goes so far as your application becomes more complex. Treating the chain of HDFS and job operations as a first class concern is where the Hadoop-specific extensions to Spring Batch come in handy. These extensions are discussed in the section "Hadoop Workflows" on page 238.

Example 11-31. Configuring the JobRunner to execute multiple HDFS scripts and jobs

```
<job-runner id="runner"
            pre-action="setupScript1,setupScript"
            job="wordcountJob1,wordcountJob2"
            post-action="cleanupScript1,cleanupScript2"/>
```

Job Scheduling

Applications often require tasks to be scheduled and executed. The task could be sending an email or running a time-consuming end-of-day batch process. *Job schedulers* are a category of software that provides this functionality. As you might expect, there is a wide range of product offerings in this category, from the general-purpose Unix cron utility to sophisticated open source and commercial products such as Quartz, Control-M, and Autosys. Spring provides several ways to schedule jobs. They include the JDK Timer, integration with Quartz, and Spring's own TaskScheduler. In this section, we will show how you can use Quartz and Spring's TaskScheduler to schedule and execute Hadoop applications.

Scheduling MapReduce Jobs with a TaskScheduler

In this example, we will add task scheduling functionality to the application developed in the previous section, which executed an HDFS script and the wordcount MapReduce job. Spring's `TaskScheduler` and `Trigger` interfaces provides the basis for scheduling tasks that will run at some point in the future. Spring provides several implementations, with common choices being `ThreadPoolTaskScheduler` and `CronTrigger`. We can use an XML namespace, in addition to an annotation-based programming model, to configure tasks to be scheduled with a trigger.

In the configuration shown in Example 11-32, we show the use of an XML namespace that will invoke a method on any Spring-managed object—in this case, the `call` method on the `JobRunner` instance. The trigger is a cron expression that will fire every 30 seconds starting on the third second of the minute.

Example 11-32. Defining a TaskScheduler to execute HDFS scripts and MapReduce jobs

```
<!-- job definition as before -->
<job id="wordcountJob" ... />

<!-- script definition as before -->
<script id="setupScript" ... />
```

```
<job-runner id="runner" pre-action="setupScript" job="wordcountJob"/>

<task:scheduled-tasks>
  <task:scheduled ref="runner" method="call" cron="3/30 * * * * ?"/>
</task:scheduled-tasks>
```

The default value of `JobRunner`'s `run-at-startup` element is `false` so in this configuration the HDFS scripts and Hadoop jobs will not execute when the application starts. Using this configuration allows the scheduler to be the only component in the system that is responsible for executing the scripts and jobs.

This sample application can be found in the directory *hadoop/scheduling*. When running the application, we can see the (truncated) output shown in Example 11-33, where the timestamps correspond to those defined by the cron expression.

Example 11-33. Output from the scheduled wordcount application

```
removing existing input and output directories in HDFS...
copying files to HDFS...
23:20:33.664 [pool-2-thread-1] WARN  o.a.hadoop.util.NativeCodeLoader
- Unable to load native-hadoop library for your platform...
23:20:33.689 [pool-2-thread-1] WARN  org.apache.hadoop.mapred.JobClient
- No job jar file set.  User classes may not be found.
23:20:33.711 [pool-2-thread-1] INFO  o.a.h.m.lib.input.FileInputFormat
- Total input paths to process : 1
23:20:34.258 [pool-2-thread-1] INFO  org.apache.hadoop.mapred.JobClient
- Running job: job_local_0001 ...
23:20:43.978 [pool-2-thread-1] INFO  org.apache.hadoop.mapred.JobClient
-  map 100% reduce 100%
23:20:44.979 [pool-2-thread-1] INFO  org.apache.hadoop.mapred.JobClient
- Job complete: job_local_0001
23:20:44.982 [pool-2-thread-1] INFO  org.apache.hadoop.mapred.JobClient
- Counters: 22
removing existing input and output directories in HDFS...
copying files to HDFS...
23:21:03.396 [pool-2-thread-1] INFO  org.apache.hadoop.mapred.JobClient
- Running job: job_local_0001
23:21:03.397 [pool-2-thread-1] INFO  org.apache.hadoop.mapred.JobClient
- Job complete: job_local_0001
```

That's it! As you can see, the task namespace is simple to use, but it also has many features (which we will not cover) related to the thread pool policies or delegation to the CommonJ `WorkManager`. The Spring Framework reference documentation describes these features in more detail.

Scheduling MapReduce Jobs with Quartz

Quartz is a popular open source job scheduler that includes many advanced features such as clustering. In this example, we replace the Spring `TaskScheduler` used in the previous example with Quartz. The Quartz scheduler requires you to define a `Job` (aka a `JobDetail`), a `Trigger`, and a `Scheduler`, as shown in Example 11-34.