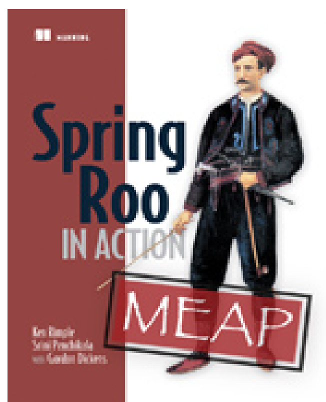


Spring Roo in Action

By Ken Rimple and Srini Penchikala with Gordon Dickens

In this green paper based on Spring Roo in Action, the authors show how Roo is accessed via the Roo Shell and then how to configure and run a web-based application. Next, they explain how Roo creates and manages your application. Finally, discuss how using Roo can simplify application code, reducing repetitive tasks.

To save 35% on your next purchase use Promotional Code **rimplegp35** when you check out at www.manning.com.

You may also be interested in…

What is Spring Roo?

You are about to be introduced to a powerful new tool that makes your life as a Java application developer less stressful and more productive. That tool is Spring Roo. Using a simple command-line shell, Roo can create and manage Spring-based applications, adding and configuring features such as JPA, Spring MVC, Web Flow, JMS, Email, testing in frameworks like Selenium, Spring Security, and much, much more. And all the while, Roo implements your configuration changes using highly optimized architecture and coding techniques and simplifies your code.

In this paper, you'll learn how Roo is accessed via the Roo Shell, and then you'll configure and run a web-based application, written in only 14 lines of Roo commands. Then we will take a look at some of the created objects to get a feel for how Roo creates and manages your application. Finally, we will wrap up by discussing how using Roo can simplify your application code, reducing the size of your code base while reducing repetitive tasks.

Let's begin by discussing one of the major reasons why Java-based development projects sputter and spin—the complexity of configuring the application architecture.

The configuration burden

Putting together a Java-based application can be an arduous task. Where do you start? There are many things to consider—build scripts, dependency management, architectural patterns, framework selections, database mapping strategies, and more.

In traditional Enterprise Java development, architects pull together a hodgepodge of open-source technologies and standards-driven platforms such as JDBC, the Servlet and JSP APIs, and Enterprise JavaBeans using a build tool such as Ant or Maven. Many architectural and design patterns can be used to make development more efficient, but most of the work involves a lot of coding, physical build configuration, and design work.

The Spring Framework, a development platform that uses interface-driven development, dependency injection, aspect-oriented programming, and a number of helper APIs and services, significantly reduces the complexity of your Enteprise Java code, but it still requires that you add dependent JAR files and configuration in order to simplify the programming tasks later. Let's illustrate this complexity with a Java Persistence framework: JPA.

Configuring a new feature—JPA

Take the case of the Java Persistence API (JPA). JPA was created to standardize the task of programming Object-Relational Mapping—developing objects that represent a data model and then mapping those objects via configuration elements such as XML or Java annotations.

There are many mapping APIs. JPA was developed to provide a common API for Java persistence, and it provides a standard set of annotations and XML which define rules to map database tables to Java objects. It also is a standard, similar to JDBC, which is implemented by a number of open source projects—such as Hibernate, EclipseLink, and Open JPA.

Let's see how much work you have to do to configure JPA on a Spring project. In this example we are using Hibernate as our JPA provider platform. We would have to:

- Include dependent JAR files in your project for the JPA API, and a number of ORM vendor JAR files, if you are using Hibernate.
- Install a JDBC data source and include the JAR files for the data source and the database driver.
- Configure a transaction management strategy in Spring to associate with the database data source and with JPA.
- Configure META-INF/persistence.xml with settings relating JPA to the database using Hibernate configuration settings.
- Configure a Spring `LocalContainerEntityManagerFactoryBean` to install a JPA container and configure the transaction management to honor the JPA Entity Manager.

That's a significant amount of work. To actually *use* the JPA API in your project, you have to also:

- Create an entity class that represents the table, annotating it with JPA annotations.
- Create a *repository* class and inject it with an `EntityManager` instance from the configuration above.
- Write methods to retrieve, save, update, delete, and search for your entities.
- Develop a Spring Service class to provide transactional services, which coordinate access to one or more calls to the Repository.

Ultimately, once your configuration is settled, you are highly productive using Spring and JPA. It's the effort of wiring up the framework that is the brunt of the work. And, if you haven't done it before, the research takes a significant amount of time.

Now, expand that selection and configuration process to your web framework, web services, email, a messaging strategy, and a rich internet client such as Google Web Toolkit, and you begin to see why it takes so long to put together a comprehensive enterprise Java application.

Too many decisions

Even when using the Spring Framework, developers are faced with a bewildering set of choices for each major architectural challenge they face (see figure 1). The more choices developers are faced with when attempting to make a decision, the more difficult that decision becomes. If the developer is less experienced, they may just begin to select a configuration option at random or start experimenting with several, comparing the results. All of this takes time and can really hold up a project.

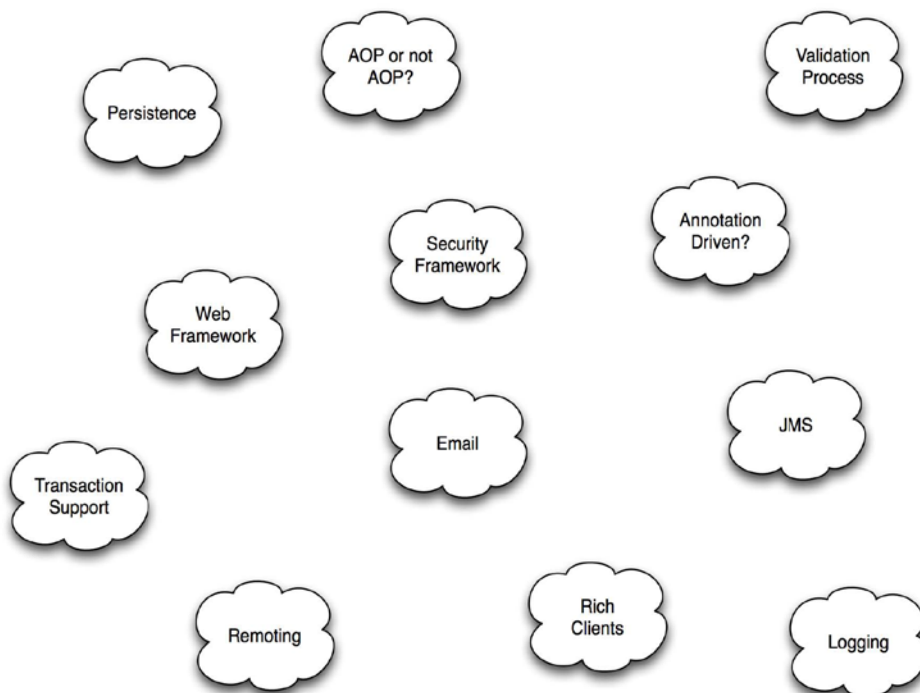


Figure 1 The number of choices when working in an Enterprise Java application is mind-numbing!

Rapid application development frameworks, such as Ruby on Rails or Grails, narrow choices and implement programming patterns in a simple yet predicable way. They also focus on removing repetitious configuration and coding from your application. As a result, they are wildly popular. However, since most of these platforms use runtime type checking and are based on interpreted languages, they aren't always suited for large development teams in the way Java projects are.

What Spring and Java need is a tool that makes it easier to configure and add features to a project and remove or hide any generated or mechanical code so you can easily locate your actual business logic.

Let's look at some of the ways developers have attempted to solve the problem of complex configuration and repetitive code on the Java platform.

CUTTING DOWN ON COMPLEXITY

Even without Roo, developers have a number of ways to speed up the process of configuring application architectures:

- *Code generation*—Generating code from data, SQL, instructions in XML, or in myriad other ways has long been a way to speed up application development. The Hibernate HBM2DDL utility, which takes Hibernate mappings and turns them into SQL commands, is used to generate database schemas from XML or annotation-driven mappings and has been a big leap forward in terms of managing an ORM solution.
- *Smart development tools*—Many IDEs such as Eclipse, SpringSource Tool Suite, NetBeans, and IntelliJ IDEA provide wizards that can configure a project from scratch or add a feature to a project, such as JPA or Spring. These are great tools but they vary from IDE to IDE, and requiring your developers to standardize on a given IDE is generally an unpopular decision.

  One big step forward in terms of fighting build complexity has been the Sonatype M2Eclipse plugin, which reduces Maven build configuration settings to a series of user-friendly property panes. Still, managing even Maven projects can take time and resources away from solving business problems.
- *Maven archetypes*—The Maven build and project management tool provides a project creation ecosystem in the form of *archetypes*—seed projects, built using text-based templates, that expand into new projects using

the maven `archetype:generate` command. This is a great way to sample a wide variety of projects, and Matt Raible's AppFuse project is a good example of how to use this feature of Maven. Eclipse, SpringSource Tool Suite, and IntelliJ now can use Maven archetypes to create new projects.

- *Spring namespace configuration*—By using Spring to begin with, you are already cutting down on your application complexity. Spring's dependency injection, third-party framework configuration support, and APIs such as templates reduce overall configuration complexity.

In recent versions of the Spring Framework, XML configuration namespaces have been created to simplify basic configuration. A one-line configuration for the Spring MVC web framework, `<mvc:annotation-driven />`, automatically sets up features such as request URL controller mappings and JSP view names. Annotation-driven Spring developers are using `<context:component-scan>` to automatically find and mount Spring Beans on startup. As Spring evolves, more of these namespaces and simple configurations will emerge.

All of these tools and approaches are helpful, but they aren't as rapid as developing on a platform like Rails or Grails, where adding a new feature is generally as simple as installing a plugin and following the conventions to use it. Modularization of features into plugins makes it easier to swap configuration later with different approaches. This is a key reason people are moving to these rapid application development platforms.

Wouldn't it be great if you could match their agility and still use Spring's highly productive APIs? What if you had a tool that was the equivalent of a seasoned architect available on demand, who could take over the job of implementing your infrastructure and architecture? What if this tool could deal with the mundane tasks of managing your application dependencies, wiring up services like JPA, email, JMS, and a database persistence layer? What if you had a virtual Spring Design Patterns expert available at your beck and call?

Effortless project configuration

Into the often tedious landscape of Java and Spring programming enters Spring Roo—a tool that acts like a really smart Spring and Java EE expert or pair programmer, who consistently nudges you in the right direction. Roo configures and manages the infrastructure of a Spring project, hides repetitive code, manages your build process, and generally helps you focus on solving your business problems.

Roo makes life easier for the average Spring programmer by managing the busy work and implementing sound architectural patterns. When you ask Roo to set up your persistence tier, it responds by adding JPA, Hibernate, transaction management, and a pooled datasource. When you ask Roo to create your first web controller, it implements Spring MVC and precreates a web framework complete with templating, navigation, and validation. When you need to send or receive JMS messages, a single command in Roo sets up your JMS listener or templates, and you can immediately get to work coding your sending or receiving bean.

From a developer's perspective, Roo pulls the tedious code out of your classes and into files that it manages for you. For example, your JPA entities will contain only the code that defines the fields, validation rules, and specific logic you require. Behind the scenes, Roo will implement the setters, getters, persistence management, and even `toString` methods in hidden files. You can choose to write these elements yourself but the advantage of doing so is eliminated for the majority of your code.

To illustrate how much Roo can help you boost your productivity on the Spring platform, let's take a quick dive into Spring Roo by running one of the sample scripts, `Vote.roo`, provided by the Roo project team.

Getting started: the Roo Shell

Roo configures and manages your application architecture using the Roo Shell. You can launch the shell either as a standalone command-line tool or as a view pane in the SpringSource Tool Suite IDE. Once launched, you interact with the shell by entering configuration commands, and Roo responds by modifying or creating the necessary files in your project.[1]

Let's get started. After installing the Spring Roo tool into your environment, open an operating system command line. Launch Roo using the `roo` command. Roo will respond with a `roo>` prompt.

---

[1] You will need to install Spring Roo on your computer.

The Roo Shell supports code completion, context sensitive help, and hinting. Try it now: hit your `TAB` key and see how Roo automatically prompts you with a list of commands you can type.[2] Try the `hint` command to ask Roo what the next logical steps are for managing your application. When you are finished experimenting with tab completion and `hint`, type `quit` to exit the Roo Shell.

We are going to use the `script` command, which executes a script provided to it by the user. Note that you can launch a single Roo Shell command by simply typing `roo` *`command`*. Let's use the `script` command to run our application configuration.

The Vote.roo sample

To assemble a simple vote tracking demo project, we will call one of Roo's built-in demonstration script files, `vote.roo`. Follow the instructions below to invoke the script within the Roo Shell, which will create the application:

1. Open your operating-system command-line shell.
2. Create a directory named `vote`.
3. Switch to the `vote` directory in your shell.
4. Type the following command: `roo script vote.roo`.

After executing the command, review the output in your command line. Notice the output contains a series of commands and Roo's responses to those commands, statements such as `Created` or `Managed`.

```
project --topLevelPackage com.springsource.vote
Created /Sandbox-Projects/vote/pom.xml
Created SRC_MAIN_JAVA
Created SRC_MAIN_RESOURCES
Created SRC_TEST_JAVA
Created SRC_TEST_RESOURCES
Created SRC_MAIN_WEBAPP
Created SRC_MAIN_RESOURCES/META-INF/spring
Created SRC_MAIN_RESOURCES/META-INF/spring/applicationContext.xml
Created SRC_MAIN_RESOURCES/META-INF/spring/log4j.properties
persistence setup --provider HIBERNATE --database HYPERSONIC_PERSISTENT
Created SRC_MAIN_RESOURCES/META-INF/persistence.xml
Created SRC_MAIN_RESOURCES/META-INF/spring/database.properties
Managed SRC_MAIN_RESOURCES/META-INF/spring/applicationContext.xml
Managed ROOT/pom.xml
...
```

If you are familiar with Maven, you'll recognize the uppercase labels such as SRC_MAIN_JAVA, which refers to the actual path of `src/main/java`. Notice that Roo actually creates configuration items for Hibernate, JPA, logging, and a number of other services automatically, based on a series of Roo Shell commands.

To get a feel for what we have built, let's run the application.

Running the vote application

To recap: in the beginning of the `vote.roo` script, Roo assembled a Maven build file,`pom.xml`, a fully configured, secured Spring MVC application, and installed JPA and Hibernate. Later in the script file, Roo also configured a web application using Tomcat and Jetty so that the project can be launched in a web container. Let's launch the application by typing `mvn tomcat:run` into the operating system command prompt. After a short wait, the application will start. Open a web browser and browse to `http://localhost:8080/vote` and you will see the page in figure 2.

[2] If launched from the SpringSource Tool Suite, Roo uses CTRL+SPACE (or CMD+SPACE on Mac)

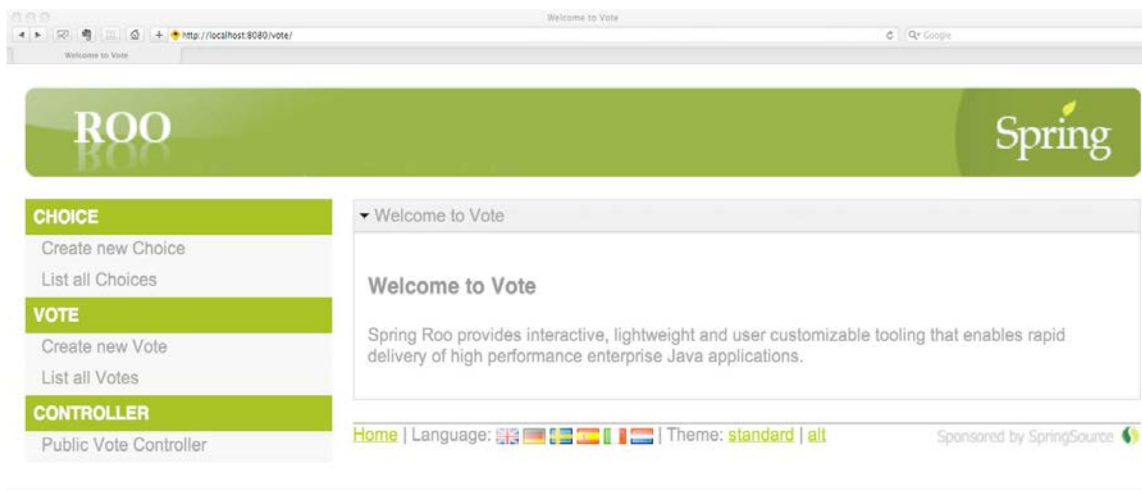


Figure 2 Your first Roo web application

After reviewing the web page, click the Create new Choice menu link. You should be directed to a login page as shown in figure 3. The vote script installed the Spring Security framework and configured the controllers as secured URLs. The links to create choices or votes will require authentication (in this demo, enter `admin` for both the username and password and click SUBMIT). Spring Security automatically redirects unsecured requests to a login page and then to the protected page.

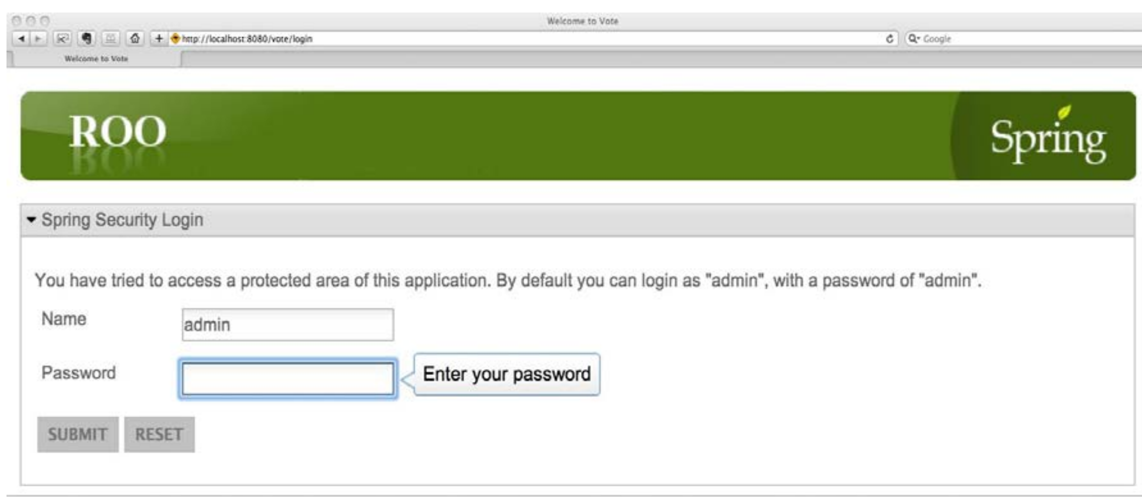


Figure 3 Intercepted by Spring Security.

Notice the tooltips that appear when you tab from field to field. These tooltips are automatically provided by the Spring Javascript API, which is installed as part of the web application.

Once you reach the Choice creation screen, click on the Naming Choice field. Try clicking on SAVE without entering a naming choice. Spring should have turned the input field yellow and marked it with a triangle warning, as depicted in figure 4.

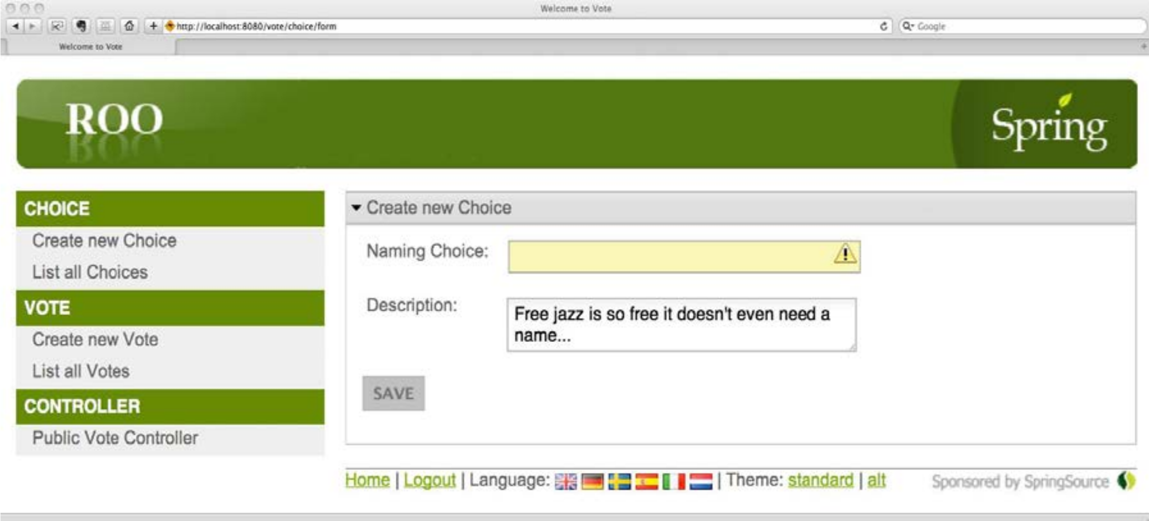


Figure 4 Required field checks as they appear graphically

Enter a few choices, perhaps country, rock and roll, jazz, and classical music. You can choose to enter a description but note that it is optional. Each time you save your choice, you are directed to a Show Choice screen to review your entered choice. You can then click on Create new Choice again to create another one or click on List all Choices to show a list of existing choices, as shown in figure 5.

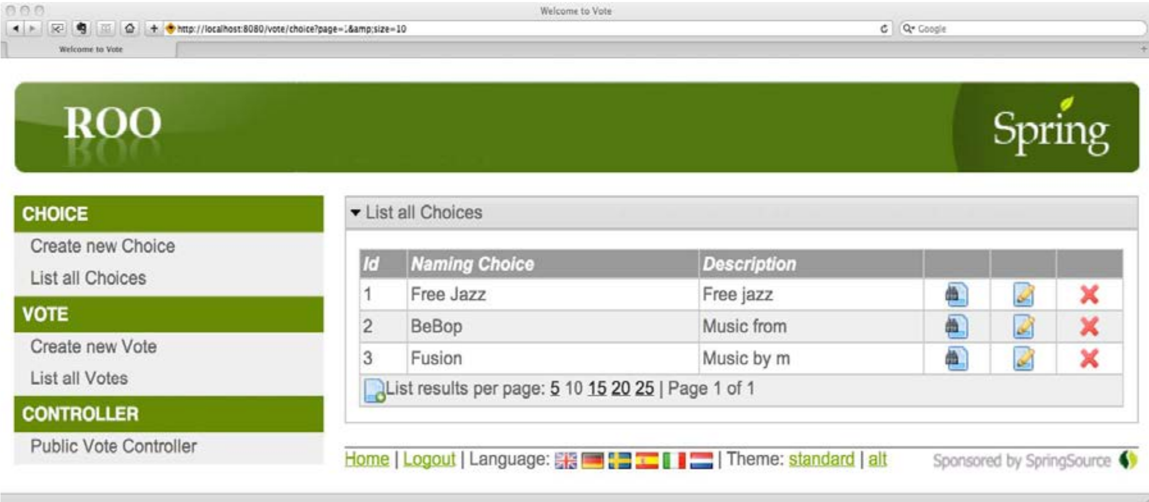


Figure 5 The list view with pagination already provided

Finally, let's record a vote. Click on the Create new Vote menu item and enter your choice. Clicking on the Choice dropdown brings up a rather ugly list made up of the fields from the Choice screen. You can streamline and customize these elements. Select a choice and enter an arbitrary IP address like `127.0.0.1` for the `Ip` field. Finally, click on the Registered input field. You'll see a pop-up calendar widget (figure 6), automatically provided by Roo. Select a date and click SAVE.

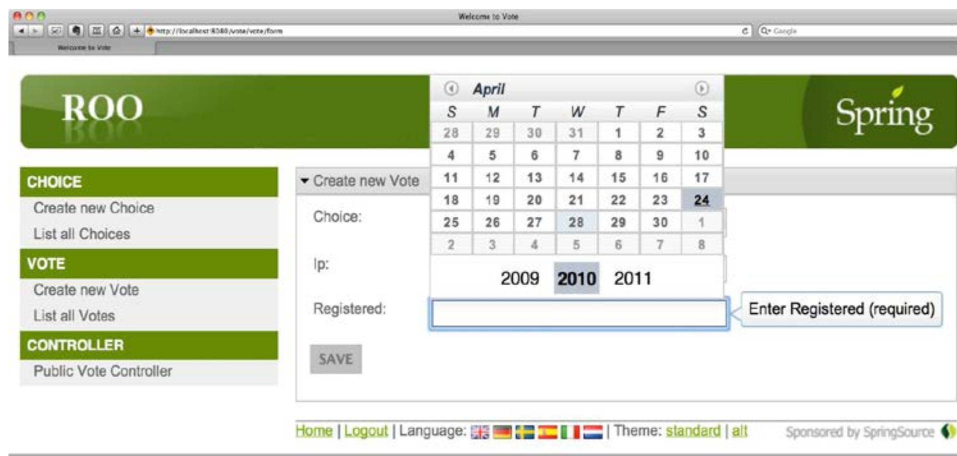


Figure 6 Recording a vote in the sample application.

You can terminate the application at this point. Go to your command window and shut down Tomcat by hitting CTRL+C. Now, let's see how Spring Roo generated our project and review some of the components that make up the voting application.

Review time

The Roo platform is built on top of several key technologies: the Roo shell, annotations, aspects, and a Maven-based Spring project platform. To better understand the interplay between these components, let's review how the Vote.roo script assembles the application we just tested. We will then discuss how Roo manages the various components in the system.

THE COMPLETE VOTE.ROO SCRIPT

This entire web entry system was generated by a few lines of Roo script, shown in listing 1.

Listing 1 The complete vote.roo script

```
project --topLevelPackage com.springsource.vote                         1

persistence setup --provider HIBERNATE à
   --database HYPERSONIC_PERSISTENT                                     2
entity --class ~.domain.Choice -testAutomatically                       3
field string namingChoice --notNull --sizeMin 1 --sizeMax 30
field string description --sizeMax 80

controller scaffold ~.web.ChoiceController                              4

entity --class Vote -testAutomatically                                  5
field reference choice --type Choice
field string ip --notNull --sizeMin 7 --sizeMax 15
field date registered --type java.util.Date --notNull --past

controller scaffold ~.web.VoteController                                6
controller class ~.web.PublicVoteController --preferredMapping /public

logging setup --level DEBUG --package WEB                               7
security setup                                                          8
```

1 Creates a new Maven project file, setting the top-level package to com.springsource.vote and creating the directories for storing our objects

2 Installs the JDBC Driver, Spring JDBC DataSource, Transaction Manager, Spring JPA Container, and the proper configuration files

3 The next three lines install a JPA entity called Choice, located in the domain sub-package, and add two fields, namingChoice and description.

4 This one line command installs the MVC controller for the Choice entity. Since it is the first one, it sets up Spring MVC, including support for annotation-driven @Controller objects, as well as support for AJAX, user interface styling using Apache Tiles.

5 This set of commands configures the Vote entity, adding a reference to the Choice entity and two additional fields: ip and registered.

6 Now we configure another MVC Controller for the Vote entity. Roo automatically adds the Vote pages to menu it has constructed for the user interface.

7 Configure an instance of log4j to support debug-level output for the SpringSource Web package structure

8 Configure Spring Security to automatically secure our website

Take special notice of `--topLevelPackage` attribute of the `project` command. This setting allows us to use the ~. shortcut in future commands as a base package to build our other components from.

Only 14 lines of Roo commands, and we have already generated a complete Spring MVC web application. The configuration includes a number of technologies:

- The Spring Framework
- Hibernate ORM
- The Java Persistence API
- Java EE Bean Validations (JSR-303)
- Spring MVC
- Spring Security
- Apache Tiles
- Logging via the SLF4J framework and LOG4J

Although Vote is a web-based, data-driven application, you may be building a message-driven system, perhaps using JMS or the Spring Integration API. Your application may be a job-scheduling system, running as a console-only program and scheduled with an API like Quartz or the new Spring Task Scheduler API. In all of these cases, Spring Roo will help you configure and manage your project, saving valuable time you would otherwise spend wiring together your infrastructure.

A closer look at the first two script entries shows how we have defined our Roo application:

```
project --topLevelPackage com.springsource.vote
persistence setup --provider HIBERNATE --database HYPERSONIC_PERSISTENT
```

The `project` command defines our top-level package (the package at the root of our project class hierarchy) as `com.springsource.vote`, which can be shortened in other Roo commands to `~.` instead. Next, the `persistence` command defines a JPA ORM configuration, backed with Hibernate, using the persistent version of the Hypersonic SQL database.

So with two short commands, we've configured the entire backend platform. Normally that would take a developer anywhere from a half-hour to a day, or more. But this script continues on to define the database entity classes and user interface elements, so let's take a look at that configuration step.

THE CHOICE JPA ENTITY

The script now configures two entity classes, `Choice` and `Vote`. Roo Entities are JPA-annotated classes that map data to a database and specify data-type constraints and validation rules using annotations such as `@NotNull`, `@Past`, and `@Size`.

The fragment used to create the `Choice` entity is defined in three Roo commands:

```
entity --class ~.domain.Choice --testAutomatically
field string namingChoice --notNull --sizeMin 1 --sizeMax 30
field string description --sizeMax 80
```

When Roo encounters the `entity` command, it creates a class file in the named location (using the special shorthand `~.` to represent the base package `com.springsource.vote`). Roo also takes command-line arguments such as `--testAutomatically`, which will actually build a JUnit test class automatically. The next two lines, starting with `field` define fields to add to the `Choice` entity. The options such as `--sizeMin` and `--sizeMax` define validation constraints that are used to drive data validation. The complete Spring Roo JPA Entity follows in listing 2.

Listing 2 The resulting `Choice.java` class

```
package com.springsource.vote.domain;
import javax.persistence.Entity;
import org.springframework.roo.addon.javabean.RooJavaBean;
import org.springframework.roo.addon.tostring.RooToString;
import org.springframework.roo.addon.entity.RooEntity;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

@Entity                                                              1
@RooJavaBean
@RooToString
@RooEntity
public class Choice {

   @NotNull                                                          2
   @Size(min = 1, max = 30)
   private String namingChoice;

   @Size(max = 80)
   private String description;                                       3

}
```

1 In addition to the JPA annotation @Entity, Roo adds @RooJavaBean, @RooToString, and @RooEntity. They are placed in the file automatically to generate repetitious code, including getters and setters, the toString() method, and all of the JPA persistence code.

2 The @NotNull and @Size attributes are Bean Validation Framework annotations. They add special validation logic to our persistence requests automatically.

3 We define our JPA fields as private member variables, and with the @RooJavaBean annotation, Roo generates getters and setters for us so we can access the properties from other classes.

Traditional JPA entities require getters, setters, and persistence logic in the form of calls to an EntityManager, usually built within a Repository or DAO class within a layered Spring architecture. Keep in mind that Roo separates mechanical, repetitive code into normally hidden files. When we create `entity`, Roo actually constructs several files. Roo uses these files to lighten the developer's load. Let's see how that can help you improve your productivity.

ROO HANDLES YOUR "LIGHT WORK"

We all know the standard line of movie *Mafia Dons* when they wanted to get rid of a troublemaker: "Take care of my light work…" How much light work do you have to do on a daily basis just to get something done? Chances are, to write the rest of the code that persists a `Choice` or a `Vote` object, quite a lot. Let's take a quick look at what Spring Roo generated behind the scenes.

Take a look at the simplified class diagram in figure 7. It shows all of the special, hidden files Roo created (with the exception of the Unit Test, which is a Java class) when we asked it to create our `Choice` object. For now, ignore the fact that they end in .aj: this is something we'll discuss later.[3]

Roo expects you to mostly ignore these files. But if you open them up in a text editor, you'll see it's the *light work* you're used to coding yourself.

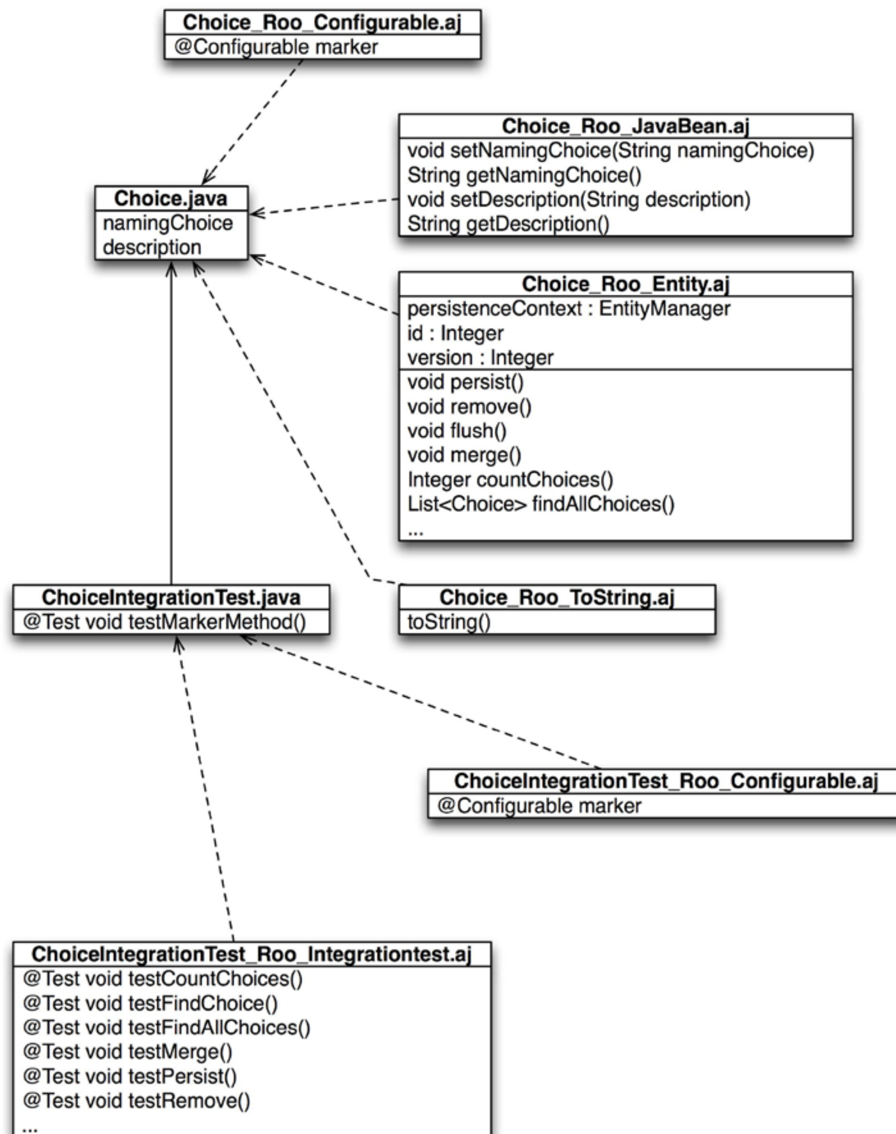


Figure 7 Related files

The `Choice` object is actually composed of several files, each adding functionality to the `Choice.java` class. These files are named with the extension `_Roo_ feature` .aj and as we will soon see, are AspectJ components called *intertype declarations* (ITDs). These files essentially hide mechanically generated implementation code.

---

[3] For the curious, Roo is one of the first projects to take advantage of intertype declarations, a feature of AspectJ.

These files are created by the Roo Shell, which takes direction from special annotations contained within the source class `Choice.java`. They are:

- `@RooEntity`—This annotation marks the class as being managed by Roo and JPA. The `Choice_Roo_Configurable.aj` and `Choice_Roo_Entity.aj` files are generated by this annotation. Persistence code is automatically written in the `Choice_Roo_Entity.aj` file, and the `Choice_Roo_Configurable.aj` file is provided to automatically configure the entity classes for JPA when they are created using the new keyword.
- `@RooJavaBean`—Generates the `Choice_Roo_JavaBean.aj` file, which adds setters and getters to any fields defined in `Choice.java`.
- `@RooToString`—Generates a convenient `toString()` method. You can rename the method or implement your own `toString()` method in the `Choice.java` file, which will remove the `Choice_Roo_ToString.aj` file.

Roo created the `ChoiceIntegrationTest` and the related ITDs based on the `--testAutomatically` flag on the `entity` command. This test actually spins up the Spring Container and runs integration tests against the entity, helping you identify problems with the database design and the mapping process early in the project lifecycle.

Roo also created the `Vote` entity using the same syntax. In addition to defining the entity itself, the script also defined a relationship between the `Vote` and `Choice` classes. This relationship was expressed in code as a JPA reference. Here is the Roo script fragment that defines the Vote class:

```
entity --class ~.domain.Vote --testAutomatically
field reference choice --type Choice
field string ip --notNull --sizeMin 7 --sizeMax 15
field date registered --type java.util.Date --notNull -past
```

The script defines the Vote entity, and three fields – `choice`, a Java reference to the `Choice` entity defined earlier, and an `ip` and `registered` field, to track the source and the date of the vote.

Listing 3 shows the Vote JPA entity.

Listing 3 The Vote Roo entity class

```
@Entity
@RooJavaBean
@RooToString
@RooEntity
public class Vote {

   @ManyToOne(targetEntity = Choice.class)                              1
   @JoinColumn
   private Choice choice;

   @NotNull                                                             2
   @Size(min = 7, max = 15)
   private String ip;

   @NotNull
   @Past
   @Temporal(TemporalType.TIMESTAMP)
   @DateTimeFormat(style = "S-")
   private Date registered;

}
```

1 Defines a relationship to the Choice Entity

2 Define annotation-driven validation rules

Roo takes a full advantage of the relationships defined between entities to build a navigable user interface by generating the dropdown list in the Vote screen and populating it with entries from the `Choice` entity. Roo uses the generated `toString` method in `@RooToString` to provide the values for the dropdown list. If you want to

create your own version, you can simply implement `toString()` in the `Choice.java` file yourself, and the value you return will appear in the dropdown.

How Roo works

How does Roo know to add support for frameworks for JPA, Bean Validation, and other frameworks? The secret is that Roo is also an application development and platform, made up of a collection of tools and APIs—the Roo Shell, Roo annotations, Maven, and AspectJ.

- *The Roo Shell*—Think of the Roo Shell as the central coordinator. When you launch the shell, it goes right to work, reviewing the state of every file under its control. Roo's shell is always watching the file system. Any additional modifications to source files will cause the shell to modify or remove the .aj files. In addition, upgrading Roo will cause it to adjust generated files once the shell is launched again. In this way, enhancements to Roo projects can take place automatically simply by upgrading the Roo installation and firing up the shell. The Roo Shell commands configure your project features, installing configuration in Spring Application Context files, adding dependencies to Maven and generating Java class files and other resources.
- *Roo Annotations*—As we've seen above, Roo annotates classes under its control with specific annotations such as `@RooToString`, `@RooEntity`, and `@RooJavaBean`. You can even add these annotations to classes you create yourself, and the Roo Shell will respond by generating the code to manage those features automatically.
- *Maven*—Roo uses Maven to configure and build your application. When the project is created via the `project` command, a maven `pom.xml` file is generated and all relevant dependencies and plugins are configured. Each time you add additional functionality, such as JMS or a Spring Web controller, the build file is adjusted. You can ask Roo to generate an Eclipse project to edit the code; simply type `mvn eclipse:eclipse` from the command-line, or from Roo type `perform eclipse`.
- *AspectJ*—At compile time, Roo uses an Aspect Oriented Programming (AOP) framework, AspectJ, to combine the code in the `Choice` class with the code in the files `Choice_Roo_Configurable.aj`, `Choice_Roo_Entity.aj`, `Choice_Roo_JavaBean.aj`, and `Choice_Roo_ToString.aj`. This is done via the AspectJ compiler, automatically configured in the project's Maven build file, `pom.xml`.

The end result is that configuration is made easier via the Roo Shell, and developers no longer need to scroll through huge files searching for the validation rules or specific user-written methods in a Roo-managed class. Let's dig a bit deeper and see what Roo Aspects look like.

AspectJ for simplification?

AspectJ (and AOP in general) is a powerful facility that lets developers weave features into their code at compile time or runtime. Generally, AspectJ developers think in terms of pointcut expressions and advices. We can assure you, unless you have an overarching reason to use AOP in your project directly, you won't have to. However, Roo takes full advantage of AspectJ and a feature known as intertype declarations to *extract* typical repetitive boilerplate code from your business class and simplify it greatly. Let's take a deeper look at one of these aspects, `Choice_Roo_ToString.aj`, shown in listing 4.

Listing 4 The `Choice_Roo_ToString.aj` file

```
package com.springsource.vote.domain;

import java.lang.String;

privileged aspect Choice_Roo_ToString {                              1

   public String Choice.toString() {                                 2
     StringBuilder sb = new StringBuilder();
     sb.append("Id: ").append(getId()).append(", ");
     sb.append("Version: ").append(getVersion()).append(", ");
     sb.append("NamingChoice: ")
       .append(getNamingChoice()).append(", ");
     sb.append("Description: ").append(getDescription());
```

```
        return sb.toString();
    }
}
```

1 AspectJ ITDs are defined using the aspect keyword instead of class

2 The method is defined using Choice.toString, which mixes the toString() into the Choice class at compile

Mixing in code at compile time through the AspectJ compiler makes any code as fast or faster as it would be when running a native Spring application. In fact, you are running a native Spring application, complete with JPA repository code, JavaBean getters and setters, and even an automatically mounted Spring Repository configuration, created each time you create the `Choice` class. AspectJ calls the extra files intertype declarations (ITDs).

Putting framework code into AspectJ ITDs solves several problems:

- Due to the removal of easily generated methods, the code in the target Roo bean is easier to read.
- The managed code can be optimized by Roo in future versions.
- The code is added at compile time to the original class and not during Spring startup time as related objects. This may reduce memory and CPU overhead, and it avoids creation of additional proxy classes.

Now let's wrap up our short tour by discussing some of the key benefits of Spring Roo.

Spring Roo == productivity

Reviewing the script that created our Vote application and the fact that you haven't even written a single line of code yet, you will begin to see the automatic productivity gains of using Roo to help construct your application. Mechanical, repetitive code and configuration is set aside and managed within system-generated files. As a developer, you can spend a lot more time focusing on your business logic.

In addition to the mechanical aspects, there are other reasons why adopting Roo as a development platform makes sense.

Leveraging the Spring ecosystem

Spring is everywhere. Projects such as ActiveMQ and Mule use it as a basis for their application architecture. Many web applications are written in Spring MVC, an API that underwent a transformation in Spring 2.5 and 3.0 and has become more convention based than configuration driven.

Spring is also a highly modular framework. For example, Roo configures all context configuration information by placing feature-specific configuration files in the `META-INF/spring`resources directory. If you install support for messaging using the `jms` command, an `applicationContext-jms.xml` configuration file is added, with the requisite settings pertaining to JMS only.

As stated before, Roo future-proofs your infrastructure. As you are configuring your application, it makes decisions based on a given *best practice*, which may improve in the future. Once you upgrade to the latest version of Roo, you can run the shell against your project again and it will automatically upgrade any code it generates to take advantage of the newer best practices.

There is another side-benefit to using Roo to configure and manage your application architecture. Because Spring Roo *is* a Spring application, you can use the Roo Shell and the generated code as a learning tool. How do you set up a JMS message listener or sender? Just install JMS from the Roo Shell and experiment with it. How about sending emails? Install the email template support. JPA/Hibernate persistence? Just set up your database and model objects in Roo and let it generate your relational database persistence code for you.

Configuring features with add-ons

Spring wraps many third-party APIs and provides easy access to them via its Factory Beans. Still, in order to take advantage of these APIs you generally have to learn both the Spring configuration as well as the base API itself, and then you must wire the configuration together by hand. This takes significant time.

Instead, let Roo configure these APIs for you. Roo uses a rich add-on system, with a community that is supporting more and more SpringSource and third party APIs every day. These add-ons are configured as additional Roo commands in the Shell, making adding dependent framework configuration easy.

Even out of the box, Roo will help you configure a number of frameworks and APIs, including:

- The Google Web Toolkit.
- Spring Web Flow.
- Spring MVC Controllers and a Spring web application, complete with templating using Tiles.
- A JMS provider and JmsTemplate.
- Selenium tests for your web interface.
- Unit or Integration testing.
- An OSGi bundle or use of other OSGi bundles, including through the OSGi OBR Bundle Repository API.
- Logging using Log4J and S.LF4J.

Spring Roo automatically configures frameworks using the proper Spring configuration and includes the proper dependencies. The time saved by this key feature alone is worth investing time in learning Roo.

Summary

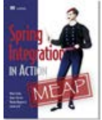
In this paper we discussed the challenge of configuring a Spring-based application and how Spring Roo can help greatly to simplify the process. We used the built-in `vote.roo` demonstration script to build a simple application, complete with security, validation, persistence, and a Spring MVC web framework, with only 14 lines of code.

We then took a quick look at how Roo manages its objects using the Roo annotations such as `@RooEntity`, `@RooJavaBean`, and `@RooToString` plus the Roo Shell and AspectJ to generate and manage repetitive code behind the scenes, simplifying your code and shining a brighter light on your business logic. Finally, we reviewed the advantages of using Spring Roo and the Spring Framework to build your applications.

Here are some other Manning titles you might be interested in:

Spring in Action, Third Edition
Craig Walls

Spring in Practice
Willie Wheeler, John Wheeler, and Joshua White



Spring Integration in Action
Mark Fisher, Jonas Partner, Marius Bogoevici, and Iwein Fuld

Last updated: November 7, 2011