Spring Roo - Reference Documentation

DISID Corporation S.L. - Pivotal Software, Inc

Table of Contents

2.0.0.M1

Preface

1. Overview

Spring Roo is an easy-to-use productivity tool for rapidly building enterprise applications in the Java programming language. It allows you to build high-quality, high-performance, lock-in-free enterprise applications in just minutes. Best of all, Roo works alongside your existing Java knowledge, skills and experience. You probably won't need to learn anything new to use Roo, as there's no new language or runtime platform needed. You simply program in your normal Java way and Roo just works, sitting in the background taking care of the things you don't want to worry about.

2. Requirements

To get started, please ensure you have the following system dependencies:

- A Linux, Apple or Windows-based operating system (other operating systems may work but are not guaranteed).
- A Java JDK 6 or newer installed. Java JDK 7 is recommended.
- Apache Maven 3.0 or above installed and in the path.

We have listed various considerations concerning the Java Development Kit (JDK) and operating systems in the known issues section of this documentation. We always recommend you use the latest version of Java and Maven that are available for your platform. We also recommend that you use Spring Tool Suite (STS) 3.7 which includes a number of features that make working with Roo even easier (you can of course use Roo with normal Eclipse or without an IDE at all if you prefer).

3. Download Spring Roo

You can download the current release from Spring Roo project page downloads section and the related documentation.

You can also build a distribution ZIP yourself from our source control repository.

4. Install Spring Roo

Once you have satisfied the initial requirements, you can install Roo by following these steps:

1. Unzip the distribution which will unpack to a single installation directory; this will be known as $ROO_HOME in the directions below
2. If using Windows, add `$ROO_HOME\bin` to your **PATH** environment variable

3. If using Linux or Apple, create a symbolic link using a command such as `sudo ln -s $ROO_HOME/bin/roo.sh /usr/bin/roo`

Next verify Roo has been installed correctly. This can be done using the following commands:

```
$ mkdir roo-test
$ cd roo-test
$ roo quit
    ____  ____  ____
   / __ \/ __ \/ __ \
  / /_/ / / / / / / /
 / _, _/ /_/ / /_/ /
/_/ |_|\____/\____/    W.X.Y.ZZ [rev RRR]

Welcome to Spring Roo. For assistance press TAB or type "hint" then hit ENTER.
$ cd ..
$ rmdir roo-test
```

If you see the logo appear, you've installed Roo successfully. For those curious, the "[rev RRR]" refers to the Git commit ID used to compile that particular build of Roo.

5. What's new in Spring Roo 2.0.0.M1

Improved extensibility

Due to the OSGi container has been upgraded to OSGi R5, now Roo provides a new way to package and distribute a set of addons together: the Roo Addon Suite.

Roo Addon Suite is based on OSGi R5 Subsystems that provides a really convenient deployment model, without compromising the modularity of Roo.

Community Addons

Now Spring Roo is centered in Spring technologies so addons like GWT addon and JSF addon have been moved to their own projects in order to be maintained by Roo community.

Feel free to contribute to Roo project by maintaining the Spring Roo Community addons.

Easier to share your addons

A place to share, to find and keep track on third party addons has been created. This place is the Roo Marketplace.

Backward compatibility

Spring Roo 2.0 has important changes to achieve its goals, due to that, it contains API changes and less

add-ons than previous version so this release is not backward compatible with 1.x.

It means Spring Roo 2.0 cannot neither update nor modify applications created with Spring Roo 1.x.

I. Getting Started

Welcome to Spring Roo! In this part of the reference guide we will explore everything you need to know in order to use Roo effectively. We've designed this part so that you can read each chapter consecutively and stop at any time. However, the more you read, the more you'll learn and the easier you'll find it to work with Roo.

Parts II, III, IV and V of this manual are more designed for reference usage and people who wish to extend Roo itself.

Chapter 1. Introduction

1.1. What is Roo?

You work with Roo by loading its "shell" in a window and leaving it running. You can interact with Roo via commands typed into the shell if you like, but most of the time you'll just go about programming in your text editor or IDE as usual. As you make changes to your project, Roo intelligently determines what you're trying to do and takes care of doing it for you automatically. This usually involves automatically detecting file system changes you've made and then maintaining files in response. We say "maintaining files" because Roo is *fully round-trip aware*. This means you can change any code you like, at any time and without telling Roo about it, yet Roo will intelligently and automatically deal with whatever changes need to be made in response. It might sound magical, but it isn't. This documentation will clearly explain how Roo works and you'll find yourself loving the approach - just like so the many other people who are already using Roo.

Before you start wondering how Roo works, let's confirm a few things it is NOT:

- *Roo is not a runtime*. Roo is not involved with your project when it runs in production. You won't find any Roo JARs in your runtime classpath or Roo annotations compiled into your classes. This is actually a wonderful thing. It means you have no lock-in to worry about (you can remove Roo from your project in just a couple of minutes!). It probably also means you won't need to get approval to use Roo (what's to approve when it's more like a command line tool than a critical runtime library like Spring Framework?). It also means there is no technical way possible for Roo to slow your project down at runtime, waste memory or bloat your deployment artefacts with JARs. We're really proud of the fact that Roo imposes no engineering trade-offs, as it was one of our central design objectives.

- *Roo is not an IDE plugin*. There is no requirement for a "Roo Eclipse plugin" or "Roo IntelliJ plugin". Roo works perfectly fine in its own operating system command window. It sits there and monitors your file system, intelligently and incrementally responding to changes as appropriate. This means you're perfectly able to use vi or emacs if you'd like (Roo doesn't mind how your project files get changed).

- *Roo is not an annotation processing library*. There is a Java 6 feature known as the annotation processing API. Roo does not use this API. This allows Roo to work with Java 5, and also gives us access to a much more sophisticated and extensible internal model.

So how does Roo actually work then? The answer to that question depends on how much detail you'd like. In super-summary form, Roo uses an add-on based architecture that performs a combination of passive and active code generation of inter-type declarations. If you're interested in how that works at a practical project level, we cover that shortly in the "Beginning With Roo: The Tutorial" chapter. Or for an advanced look at Roo internals, we've covered that in Part III: Internals and Add-On Development.

1.2. Why Use It

There are dozens of reasons people like to use Roo. We've worked hard to make it an attractive tool that delivers real value without imposing unpleasant trade-offs. Nonetheless, there are five major reasons why people like Roo and use it. Let's discuss these major reasons below.

1.2.1. Higher Productivity

With Roo it is possible for Java developers to build sophisticated enterprise applications in a best-practice manner within minutes. This is not just a marketing claim, but it's a practical fact you can experience yourself by trying the ten minute test.

Anyone who has programmed Java for a few years and looked at the alternatives on other platforms will be fully aware that enterprise Java suffers from productivity problems. It takes days to start a new project and incredibly long cycle times as you go about normal development. Still, we remain with Java because it's a highly attractive platform. It's the most widely used programming language on the planet, with millions of competent developers. It has first-class tooling, excellent runtime performance, numerous mature libraries and widely-supported standards. Java is also open source, has multiple vendors and countless choice.

We built Roo because we want enterprise Java developers to enjoy the same productivity levels that developers on other platforms take for granted. Thanks to Roo, Java developers can now enjoy this higher productivity *plus* a highly efficient, popular, scalable, open, reliable platform. Best of all, in five years time it will still be possible to hire millions of people who can look at those Roo-based projects and understand what is going on and maintain them (even if you've stopped using Roo by then).

Roo's higher productivity is provided both at original project creation, and also as a developer builds out the rest of the project. Because Roo provides round-trip support, the higher productivity is automatically provided over the full lifespan of a project. This is particularly important given the long-term maintenance costs of a project far outweigh the initial development costs. While you can use Roo just for an initial jump-start if you so wish, your return on investment is exponential as you continue using it throughout a project lifespan.

Finally, while individual productivity is important, most of us work in teams and know that someday someone else will probably maintain the code we've written. As professionals we follow architectural standards and conventions to try and ensure that our present and future colleagues will be able to understand what we did, why, and have an easy time maintaining it. Our organisations often establish standards for us to follow in an effort to ensure other projects are tackled in similar ways, thus allowing people to transfer between projects and still be productive. Of course, most organisations also have people of greatly differing backgrounds and experiences, with new graduates typically working alongside more experienced developers and architect-level experts. Roo helps significantly in this type of real-world environment because it automatically implements specific design patterns in an optimal convention-over-configuration manner. This ensures consistency of implementation within a given Roo-based project, as well as across all other Roo-based projects within an organisation (and even outside your organisation, which greatly helps with hiring). Of course, the fact Roo builds on stock-

standard Java also means people of vastly different experience levels can all be highly productive and successful with Roo.

1.2.2. Stock-Standard Java

It's no longer necessary to switch platform or language to achieve extremely high levels of productivity! We designed Roo from the outset so those people with existing Java 5 knowledge, skills and experience would feel right at home. If you've ever built an enterprise application with Java, some or all of the technologies that Roo uses by default will already be familiar to you.

Some of the common technologies Roo projects use include Spring (such as Spring Framework and Spring Security), Maven, Java Server Pages (JSP), Java Persistence API (JPA, such as Hibernate), Tiles and AspectJ. We've chosen technologies which are extremely commonly used in enterprise Java projects, ensuring you've probably either already used them or at least will have no difficulty finding hundreds of thousands of other people who have (and the resultant books, blogs, samples etc that exist for each). Also, because most of these technologies are implemented using add-ons, if you'd like Roo to use a different technology on your project it's quite easy to do so.

By using standard Java technologies, Roo avoids reinventing the wheel or providing a limited-value abstraction over them. The technologies are available to you in their normal form, and you can use them in the same way as you always have. What Roo brings to the table is automatic setup of those technologies into a Spring-certified best-practice application architecture and, if you wish, automatic maintenance of all files required by those technologies (such as XML, JSP, Java etc). You'll see this in action when you complete the ten minute test.

You'll also find that Roo adopts a very conservative, incremental approach to adding technologies to your project. This means when you first start a new project Roo will only assume you want to build a simple JAR. As such it will have next to no dependencies. Only when you ask to add a persistence provider will JPA be installed, and only when you add a field using JavaBean Validation annotations will that library be installed. The same holds true for Spring Security and the other technologies Roo supports. With Roo you really do start small and incrementally add technologies if and when you want to, which is consistent with Roo's philosophy of there being no engineering trade-offs.

1.2.3. Usable and Learnable

There are many examples of promising technologies that are simply too hard for most people to learn and use. With Roo we were inspired by the late Jef Raskin's book, "The Humane Interface". In the book Raskin argued we have a duty to make things so easy to use that people naturally "habituate" to the interface, that text-based interfaces are often more appropriate than GUIs, and that your "locus of attention" is all that matters to you and a machine should never disrupt your locus of attention and randomly impose its idiosyncratic demands upon you.

With Roo we took these ideas to heart and designed a highly usable interface that lets you follow your locus of attention. This means you can do things in whatever order you feel is appropriate and never be subservient to the Roo tool. You want to delete a file? Just do it. You want to edit a file? Just do it. You want to change the version of Spring you're using? Just do it. You want to remove Roo? Just do it. You

want to hand-write some code Roo was helping you with? Just do it. You want to use Emacs and Vim at the same time? No problem. You forgot to load Roo when you were editing some files? That's no problem either (in fact you can elect to never load Roo again and your project will remain just fine).

Because Roo uses a text-based interface, there is the normal design trade-off between learnability, expressability and conciseness. No text-based interface can concurrently satisfy all three dimensions. With Roo we decided to focus on learnability and expressability. We decided conciseness was less important given the Roo shell would provide an intuitive, tab-based completion system. We also added other features to deliver conciseness, such as contextual awareness (which means Roo determines the target of your command based on the command completed before it) and command abbreviation (which means you need only type in enough of the command so Roo recognises what you're trying to do).

The learnability of Roo is concurrently addressed on three fronts. First, we favor using standard Java technologies that you probably already know. Second, we are careful to keep Roo out of your way. The more Roo simply works in the background automatically without needing your involvement, the less you *need* to learn about it in the first place. This is consistent with Raskin's recommendation to never interrupt your locus of attention. Third, we offer a lot of learnability features in Roo itself. These include the "hint" command, which suggests what you may wish to do next based on your present project's state. It's quite easy to build an entire Roo project simply by typing "hint", pressing enter, and following the instructions Roo presents (we do this all the time during conference talks; it's always easier than remembering commands!). There's also the intelligent tab completion, which has natural, friendly conventions like completing all mandatory arguments step-by-step (without distracting you with unnecessary optional arguments). There's also the online "help" command, sample scripts, this documentation and plenty of other resources.

Roo also follows a number of well-defined conventions so that you always know what it's doing. Plus it operates in a "fail safe" manner, like automatically undoing any changes it makes to the file system should something go wrong. You'll quickly discover that Roo is a friendly, reliable companion on your development journey. It doesn't require special handling and it's always there for you when you need it.

In summary, we've spent a lot of time thinking about usability and learnability to help ensure you enjoy your Roo experience.

1.2.4. No Engineering Trade-Offs

Roo doesn't impose any engineering trade-offs on your project. In fact, compared with most Spring-based enterprise applications, we're almost certain you'll find a Roo application will have a smaller deployment artefact, operate more quickly in terms of CPU time, and consume less memory. You'll also find you don't miss out on any of the usual IDE services like code assist, debugging and profiling. We'll explore how Roo achieves this below, but this information is relatively advanced and is provided mainly for architects who are interested in Roo's approach. As this knowledge is *not* required to simply use Roo, feel free to jump ahead to the next section if you wish.

Smaller deployment artefacts are achieved due to Roo's incremental dependency addition approach.

You start out with a small JAR and then we add dependencies only if you actually need them. As of Roo 1.0.0, a typical Roo-based web application WAR is around 13 Mb. This includes major components like Spring, Spring JavaScript (with embedded Dojo) and Hibernate, plus a number of smaller components like URL rewriting. As such Roo doesn't waste disk space or give you 30+ Mb WARs, which results in faster uploads and container startup times.

Speaking of startup times, Roo uses AspectJ's excellent compile-time weaving approach. This gives us a lot more power and flexibility than we'd ordinarily have, allowing us to tackle advanced requirements like advising domain objects and dependency injecting them with singletons. It also means the dynamic proxies typically created when loading Spring are no longer required. Roo applications therefore startup more quickly, as there's no dynamic proxy creation overhead. Plus Roo applications operate more quickly, as there's no dynamic proxy objects adding CPU time to the control flow.

Because Roo's AspectJ usage means there are no proxy objects, you also save the memory expense of having to hold them. Furthermore, Roo has no runtime component, so you won't lose any memory or CPU time there either. Plus because Roo applications use Java as their programming language, there won't be any classes being created at runtime. This means a normal Roo application won't suffer exhaustion of permanent generation memory space.

While some people would argue these deployment size, CPU and memory considerations are minor, the fact is they add up when you have a large application that needs to scale. With Roo your applications will use your system resources to their full potential. Plus as we move more and more enterprise applications into virtualized and cloud-hosted environments, the requirement for performant operation on shared hardware will become even more relevant.

You'll also find that Roo provides a well thought out application architecture that delivers pragmatism, flexibility and ease of maintainability. You'll see we've made architectural decisions like eliminating the DAO layer, using annotation-based dependency injection, and automatically providing dependency injection on entities. These decisions dramatically reduce the amount of Java and XML code you have to write and maintain, plus improve your development cycle times and refactoring experiences.

With Roo, you don't have to make a trade-off between productivity or performance. Now it's easy to have both at the same time.

1.2.5. Easy Roo Removal

One of the biggest risks when adopting a new tool like Roo is the ease at which you can change your mind in the future. You might decide to remove a tool from your development ecosystem for many different reasons, such as changing requirements, a more compelling alternative emerging, the tool having an unacceptable number of bugs, or the tool not adequately supporting the versions of other software you'd like to use. These risks exist in the real world and it's important to mitigate the consequences if a particular tool doesn't work out in the long-term.

Because Roo does not exist at runtime, your risk exposure from using Roo is already considerably diminished. You can decide to stop using Roo and implement that decision without even needing to change any production deployment of the application.

If you do decide to stop using Roo, this can be achieved in just a few minutes. There is no need to write any code or otherwise make significant changes. We've covered the short removal process in a dedicated removing Roo chapter, but in summary you need to perform a "push in refactor" command within Eclipse and then do a quick regular expression-based find and replace. That's all that is needed to 100% remove Roo from your project. We often remove Roo from a project during conference demonstrations just to prove to people how incredibly easy it is. It really only takes two to three minutes to complete.

We believe that productivity tools should earn their keep by providing you such a valuable service that you *want*to continue using them. We've ensured Roo will never lock you in because (a) it's simply the right and credible thing to do engineering-wise and (b) we want Roo to be such an ongoing help on your projects that you actually *choose*to keep it. If you're considering alternative productivity tools, consider whether they also respect your right to decide to leave and easily implement that decision, or if they know you're locked in and can't do much about it.

Chapter 2. Beginning With Roo: The Tutorial

In this chapter we'll build an app step-by-step together in a relatively fast manner so that you can see how to typically use Roo in a normal project. We'll leave detailed features and side-steps to other sections of this manual.

2.1. What You'll Learn

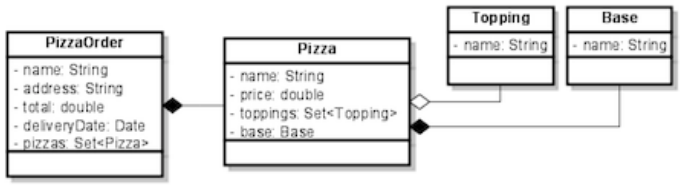
In this tutorial you will learn to create a complete Web application from scratch using Roo. The application we are going to develop will demonstrate many of the core features offered by Roo. In particular you will learn how to use the Roo shell for:

- project creation
- creation and development of domain objects (JPA entities)
- adding fields of different types to the domain objects
- creating relationships between domain objects
- automatic creation of integration tests
- creating workspace artifacts to import the project into your IDE
- automatic scaffolding of a Web tier
- running the application in a Web container
- controlling and securing access to different views in the application
- customizing the look and feel of the Web UI for our business domain
- creating and running Selenium tests
- deployment and backup of your application

2.2. Tutorial Application Details

To demonstrate the development of an application using Spring Roo we will create a Web site for a Pizza Shop. The requirements for the Roo Pizza Shop application include the ability to create new Pizza types by the staff of the Roo Pizza Shop. A pizza is composed of a base and one or more toppings. Furthermore, the shop owner would like to allow online orders of Pizzas by his customers for delivery.

After this short discussion with the Pizza Shop owner, we have created a simple class diagram for the initial domain model:



While this class diagram represents a simplified model of the problem domain for the pizza shop problem domain, it is a good starting point for the project at hand in order to deliver a first prototype of the application to the Pizza Shop owner. Later tutorials will expand this domain model to demonstrate more advanced features of Spring Roo.

2.3. Step 1: Starting a Typical Project

Now that we have spoken with our client (the Pizza Shop owner) to gather the first ideas and requirements for the project we can get started with the development of the project. After installing a JDK, Spring Roo and Maven, we create a new directory for our project:

```
> mkdir pizza
> cd pizza
pizza>
```

Next, we start Spring Roo and type '**hint**' to obtain context-sensitive guidance from the Roo shell:

```
pizza> roo
    ____  ____  ____
   / __ \/ __ \/ __ \
  / /_/ / / / / / / /
 / _, _/ /_/ / /_/ /
/_/ |_|\____/\____/    1.2.1.RELEASE [rev 6eae723]


Welcome to Spring Roo. For assistance press TAB or type "hint" then hit ENTER.
roo>
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://stackoverflow.com/questions/tagged/spring-roo
for Roo help.
roo>
```

There are quite a few usability features within the Roo shell. After typing **hint** you may have noticed that this command guides you in a step-by-step style towards the completion of your first project. Or if you type **help** you will see a list of all commands available to you in the particular context you are in. In our case we have not created a new project yet so the help command only reveals higher level commands which are available to you at this stage. To create an actual project we can use the **project** command:

```
roo> project setup --topLevelPackage com.springsource.roo.pizzashop
Created ROOT/pom.xml
Created SRC_MAIN_RESOURCES
Created SRC_MAIN_RESOURCES/log4j.properties
Created SPRING_CONFIG_ROOT
Created SPRING_CONFIG_ROOT/applicationContext.xml
com.springsource.roo.pizzashop roo>
```

When you used the project command, Roo created you a Maven `pom.xml` file as well as a Maven-style directory structure. The top level package you nominated in this command was then used as the `<groupId>` within the `pom.xml`. When typing later Roo commands, you can use the "~" shortcut key to

refer to this top-level-package (it is read in by the Roo shell from the `pom.xml` each time you load Roo).

The following folder structure now exists in your file system:

For those familiar with Maven you will notice that this folder structure follows standard Maven conventions by creating separate folders for your main project resources and tests. Roo also installs a default application context and a log4j configuration for you. Finally, the project pom file contains all required dependencies and configurations to get started with our Pizza Shop project.

Once the project structure is created by Roo you can go ahead and install a persistence configuration for your application. Roo leverages the Java Persistence API (JPA) which provides a convenient abstraction to achieve object-relational mapping. JPA takes care of mappings between your persistent domain objects (entities) and their underlying database tables. To install or change the persistence configuration in your project you can use the **jpa setup** command (note: try using the <**TAB**> as often as you can to auto-complete your commands, options and even obtain contextual help):

```
com.springsource.roo.pizzashop roo> hint
Roo requires the installation of a persistence configuration,
for example, JPA.

For JPA, type 'jpa setup' and then hit TAB three times.
We suggest you type 'H' then TAB to complete "HIBERNATE".
After the --provider, press TAB twice for database choices.
For testing purposes, type (or TAB) HYPERSONIC_IN_MEMORY.
If you press TAB again, you'll see there are no more options.
As such, you're ready to press ENTER to execute the command.

Once JPA is installed, type 'hint' and ENTER for the next suggestion.
com.springsource.roo.pizzashop roo>
com.springsource.roo.pizzashop roo> jpa setup --provider HIBERNATE --database
HYPERSONIC_IN_MEMORY
Created SPRING_CONFIG_ROOT/database.properties
Updated SPRING_CONFIG_ROOT/applicationContext.xml
Created SRC_MAIN_RESOURCES/META-INF/persistence.xml
Updated ROOT/pom.xml [added dependencies org.hsqldb:hsqldb:1.8.0.10,
org.hibernate:hibernate-core:3.6.9.Final,
org.hibernate:hibernate-entitymanager:3.6.9.Final,
org.hibernate.javax.persistence:hibernate-jpa-2.0-api:1.0.1.Final,
org.hibernate:hibernate-validator:4.2.0.Final, javax.validation:validation-api:1.0.0.GA,
cglib:cglib-nodep:2.2.2,
javax.transaction:jta:1.1, org.springframework:spring-jdbc:${spring.version},
org.springframework:spring-orm:${spring.version}, commons-pool:commons-pool:1.5.6,
commons-dbcp:commons-dbcp:1.3]
com.springsource.roo.pizzashop roo>
```

So in this case we have installed Hibernate as the object-relational mapping (ORM)-provider. Hibernate is one of ORM providers which Roo currently offers. EclipseLink, OpenJPA, and DataNucleus represent the alternative choices. In a similar fashion we have chosen the Hypersonic in-memory database as our target database. Hypersonic is a convenient database for Roo application development because it relieves the developer from having to install and configure a production scale database.

When you are ready to test or install your application in a production setting, the **jpa setup** command can be repeated. This allows you to nominate a different database, or even ORM. Roo offers TAB completion for production databases including Postgres, MySQL, Microsoft SQL Server, Oracle, DB2, Sybase, H2, Hypersonic and more. Another important step is to edit the `SRC_MAIN_RESOURCES/META-INF/persistence.xml` file and modify your JPA provider's DDL (schema management) configuration setting so it preserves the database between restarts of your application. To help you with this, Roo automatically lists the valid settings for your JPA provider as a comment in that file. Note that by default your JPA provider will drop all database tables each time it reloads. As such you'll certainly want to change this setting.

Please note: The Oracle and DB2 JDBC drivers are not available in public maven repositories. Roo will install standard dependencies for these drivers (if selected) but you may need to adjust the version number or package name according to your database version. You can use the following maven command to install your driver into your local maven repository: `mvn install:install-file -DgroupId=com.oracle -DartifactId=ojdbc14 -Dversion=10.2.0.2 -Dpackaging=jar -Dfile=/path/to/file` (example for the Oracle driver)

2.4. Step 2: Creating Entities and Fields

Now it is time to create our domain objects and fields which we have identified in our class diagram. First, we can use the **entity jpa** command to create the actual domain object. The entity jpa command has a number of optional attributes and one required attribute which is `--class`. In addition to the required `--class` attribute we use the `--testAutomatically` attribute which conveniently creates integration tests for a domain object. So let's start with the `Topping` domain object:

```
com.springsource.roo.pizzashop roo> hint
You can create entities either via Roo or your IDE.
Using the Roo shell is fast and easy, especially thanks to the TAB completion.

Start by typing 'ent' and then hitting TAB twice.
Enter the --class in the form '~.domain.MyEntityClassName'
In Roo, '~' means the --topLevelPackage you specified via 'create project'.

After specify a --class argument, press SPACE then TAB. Note nothing appears.
Because nothing appears, it means you've entered all mandatory arguments.
However, optional arguments do exist for this command (and most others in Roo).
To see the optional arguments, type '--' and then hit TAB. Mostly you won't
need any optional arguments, but let's select the --testAutomatically option
and hit ENTER. You can always use this approach to view optional arguments.

After creating an entity, use 'hint' for the next suggestion.
com.springsource.roo.pizzashop roo>
com.springsource.roo.pizzashop roo> entity jpa --class ~.domain.Topping
--testAutomatically
Created SRC_MAIN_JAVA/com/springsource/roo/pizzashop/domain
Created SRC_MAIN_JAVA/com/springsource/roo/pizzashop/domain/Topping.java
Created SRC_TEST_JAVA/com/springsource/roo/pizzashop/domain
Created SRC_TEST_JAVA/com/springsource/roo/pizzashop/domain/ToppingDataOnDemand.java
Created SRC_TEST_JAVA/com/springsource/roo/pizzashop/domain/ToppingIntegrationTest.java
Created SRC_MAIN_JAVA/com/springsource/roo/pizzashop/domain/Topping_Roo_Configurable.aj
Created SRC_MAIN_JAVA/com/springsource/roo/pizzashop/domain/Topping_Roo_ToString.aj
Created SRC_MAIN_JAVA/com/springsource/roo/pizzashop/domain/Topping_Roo_Jpa_Entity.aj
Created
SRC_MAIN_JAVA/com/springsource/roo/pizzashop/domain/Topping_Roo_Jpa_ActiveRecord.aj
Created
SRC_TEST_JAVA/com/springsource/roo/pizzashop/domain/ToppingDataOnDemand_Roo_Configurable.
aj
Created
SRC_TEST_JAVA/com/springsource/roo/pizzashop/domain/ToppingDataOnDemand_Roo_DataOnDemand.
aj
Created
SRC_TEST_JAVA/com/springsource/roo/pizzashop/domain/ToppingIntegrationTest_Roo_Configurab
le.aj
Created
SRC_TEST_JAVA/com/springsource/roo/pizzashop/domain/ToppingIntegrationTest_Roo_Integratio
nTest.aj
```

You will notice that besides the creation of Java and AspectJ sources, the **entity jpa** command in the Roo shell takes care of creating the appropriate folder structure in your project for the top level package you defined earlier. You will notice that we used the '~' character as a placeholder for the project's top level package. While this serves a convenience to abbreviate long commands, you can also

tab-complete the full top level package in the Roo shell.

As a next step we need to add the 'name' field to our `Topping` domain class. This can be achieved by using the **field** command as follows:

```
~.domain.Topping roo> hint
You can add fields to your entities using either Roo or your IDE.

To add a new field, type 'field' and then hit TAB. Be sure to select
your entity and provide a legal Java field name. Use TAB to find an entity
name, and '~' to refer to the top level package. Also remember to use TAB
to access each mandatory argument for the command.

After completing the mandatory arguments, press SPACE, type '--' and then TAB.
The optional arguments shown reflect official JSR 303 Validation constraints.
Feel free to use an optional argument, or delete '--' and hit ENTER.

If creating multiple fields, use the UP arrow to access command history.

After adding your fields, type 'hint' for the next suggestion.
To learn about setting up many-to-one fields, type 'hint relationships'.
~.domain.Topping roo>
~.domain.Topping roo> field string --fieldName name --notNull --sizeMin 2
Updated SRC_MAIN_JAVA/com/springsource/roo/pizzashop/domain/Topping.java
Updated
SRC_TEST_JAVA/com/springsource/roo/pizzashop/domain/ToppingDataOnDemand_Roo_DataOnDemand.
aj
Created SRC_MAIN_JAVA/com/springsource/roo/pizzashop/domain/Topping_Roo_JavaBean.aj
```

As explained in the documentation by typing the **hint** command you can easily add constraints to your fields by using optional attributes such as `--notNull` and `--sizeMin 2`. These attributes result in standards-compliant JSR-303 annotations which Roo will add to your field definition in your Java sources. You will also notice that the Roo shell is aware of the current context within which you are using the **field** command. It knows that you have just created a Topping entity and therefore assumes that the field command should be applied to the Topping Java source. Roo's current context is visible in the shell prompt.

If you wish to add the field to a different target type you can specify the `--class` attribute as part of the **field** command which then allows you to tab complete to any type in your project.

As a next step you can create the `Base` and the `Pizza` domain object in a similar fashion by issuing the following commands (shell output omitted):

```
entity jpa --class ~.domain.Base --testAutomatically
field string --fieldName name --notNull --sizeMin 2
entity jpa --class ~.domain.Pizza --testAutomatically
field string --fieldName name --notNull --sizeMin 2
field number --fieldName price --type java.lang.Float
```

After adding the name and the price field to the `Pizza` domain class we need to deal with its relationships to `Base` and `Topping`. Let's start with the m:m (one `Pizza` can have many `Toppings` and one `Topping` can be applied to many `Pizzas`) relationship between `Pizza` and `Toppings`. To create such many-to-many relationships Roo offers the **field set** command:

```
~.domain.Pizza roo> field set --fieldName toppings --type ~.domain.Topping
```

As you can see it is easy to define this relationship even without knowing about the exact JPA annotations needed to create this mapping in our `Pizza` domain entity. In a similar way you can define the m:1 relationship between the `Pizza` and `Base` domain entities by using the **field reference** command:

```
~.domain.Pizza roo> field reference --fieldName base --type ~.domain.Base
```

In a similar fashion we can then continue to create the `PizzaOrder` domain object and add its fields by leveraging the **field date** and **field number** commands:

```
entity jpa --class ~.domain.PizzaOrder --testAutomatically
field string --fieldName name --notNull --sizeMin 2
field string --fieldName address --sizeMax 30
field number --fieldName total --type java.lang.Float
field date --fieldName deliveryDate --type java.util.Date
field set --fieldName pizzas --type ~.domain.Pizza
```

This concludes this step since the initial version of the domain model is now complete.

2.5. Step 3: Integration Tests

Once you are done with creating the first iteration of your domain model you naturally want to see if it works. Luckily we have instructed Roo to create integration tests for our domain objects all along. Hint: if you have not created any integration tests while developing your domain model you can still easily create them using the **test integration** command. Once your tests are in place it is time to run them using the **perform tests** command:

```
~.domain.PizzaOrder roo> perform tests
...
-------------------------------------------------------
 T E S T S
-------------------------------------------------------

Tests run: 36, Failures: 0, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 3.860s
[INFO] Finished at: Tue Feb 14 18:01:45 EST 2012
[INFO] Final Memory: 6M/81M
[INFO] ------------------------------------------------------------------------
```

As you can see Roo has issued a Maven command (equivalent to running 'mvn test' outside the Roo shell) in order to execute the integration tests. All tests have passed, Roo has generated 9 integration tests per domain object resulting in a total of 36 integration tests for all 4 domain objects.

2.6. Step 4: Using Your IDE

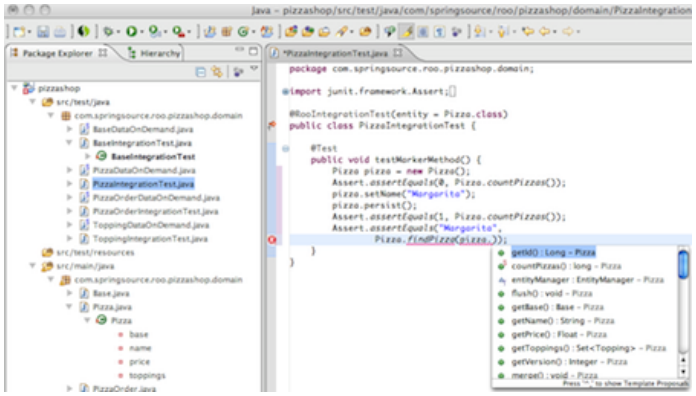
Of course Roo projects can be used in your favorite IDE. We recommend the use of SpringSource Tool Suite (STS), which is available at no charge from SpringSource. If you're not using SpringSource Tool Suite, please refer to the IDE usage section of this reference guide for a more detailed discussion of IDE interoperability.

Roo creates projects that follow standard Maven conventions. This means your IDE will be able to import your Pizza Shop project, just import it as any other Maven project. If you're an STS user, you can import your project into STS by clicking 'File > Import > Maven > Existing Maven Projects'.

Then select the new project and build it by executing 'Project > Clean ...'.

Once your project is imported and built you can take a look at the Java sources. For example you can run the included JUnit tests by right clicking the pizzashop project and then selecting 'Run As > JUnit Test'.

As detailed in the Application Architecture chapter of this documentation Roo projects leverage AspectJ Intertype declarations extensively. This does not, however, affect your ability to use code completion features offered by STS. To see code completion working in action you can open an existing integration test and use the `testMarkerMethod()` method to test it. For example you can open the `BaseIntegrationTest.java` source file and try it out:



Note, most of the methods visible in the STS code assist are actually not in the Java sources but rather part of the AspectJ ITD and are therefore introduced into the Java bytecode at compile time.

2.6.1. Edit, modify and customize the Roo-generated code

You can easily modify the Roo-generated code by using AJDT Refactoring Push-in feature.

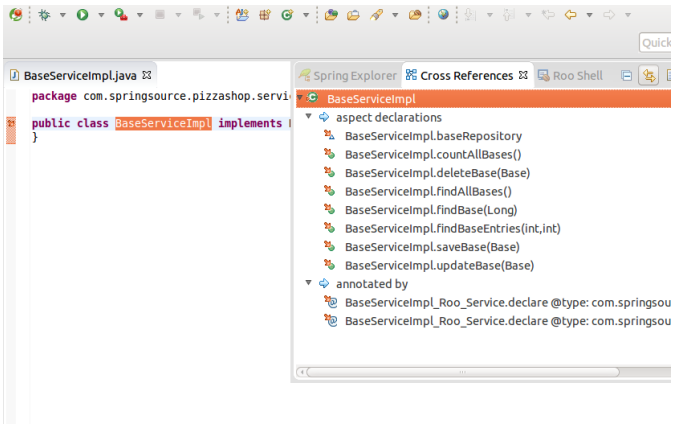
As detailed in the Critical Technologies chapter, the AJDT refactoring moves intertype declarations (methods, fields, etc) into their target types. From then, the method, field, etc. will be in the Java source file. Roo detects that change in the project and the declaration in the Java file will take priority over code generation so Roo won't re-generate it whereas the declaration is in the Java file.

To *push-in* the Roo-generated code:

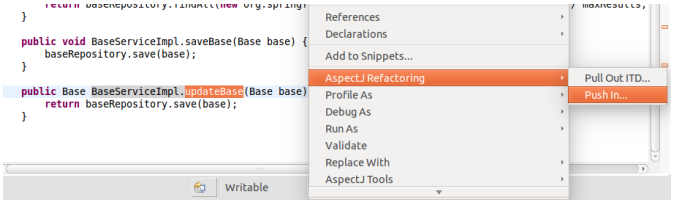
1. Edit Java source file.
2. Open the Cross References view.

TIP

If the Cross References view appears empty you must re-build the project by executing 'Project > Clean ...'. It occurs when the crosscutting information is missing, so you must re-build the project in order to re-generate the crosscutting information shown in the Cross References view.



3. Double click on the aspect declaration. The the ITD file is opened in the AspectJ/Java editor.
4. Right click ont he aspect declaration, then run 'AspectJ Refactoring > Push In ...'.



5. Finally re-build the project by executing 'Project > Clean ...'

From there, the developer can modify the Java source file directly, Roo will not neither overwrite nor modify any Java source file, they are available for customization by you from that point forward.

2.7. Step 5: Creating A Web Tier

As a next step we want to scaffold a Web tier for the Pizza Shop application. This is accomplished via the *web mvc* commands. The most convenient way to generate controllers and all relevant Web artifacts is to use the web mvc setup command followed by the web mvc all command:

```
~.domain.PizzaOrder roo> web mvc setup

~.domain.PizzaOrder roo> web mvc all --package ~.web
```

This command will scan the Pizza Shop project for any domain entities and scaffold a Spring MVC controller for each entity detected. The `--package` attribute is needed to specify in which package the controllers should be installed. This command can be issued from your normal Roo shell or from the Roo shell, which ships with STS. In order to use the integrated Roo shell within STS you need to right click on the pizzashop application and select 'Spring Tools > Open Roo Shell'.

Note, that with the web mvc setup command the nature of the project changes from a normal Java project nature to a Web project nature in STS. This command will also add additional dependencies such as Spring MVC, Tiles, etc to your project. In order to update the project classpath within STS with these new dependencies you can issue 'perform eclipse' again, followed by a project refresh in STS.

All newly added Web artifacts which are needed for the view scaffolding can be found under the `src/main/webapp` folder. This folder includes graphics, cascading style sheets, Java Server pages, Tiles configurations and more. The purpose of these folders is summarized in the UI customization section. The Roo generated Spring MVC controllers follow the REST pattern as much as possible by leveraging new features introduced with the release of Spring Framework v3. The following URI - Resource mappings are applied in Roo generated controllers:

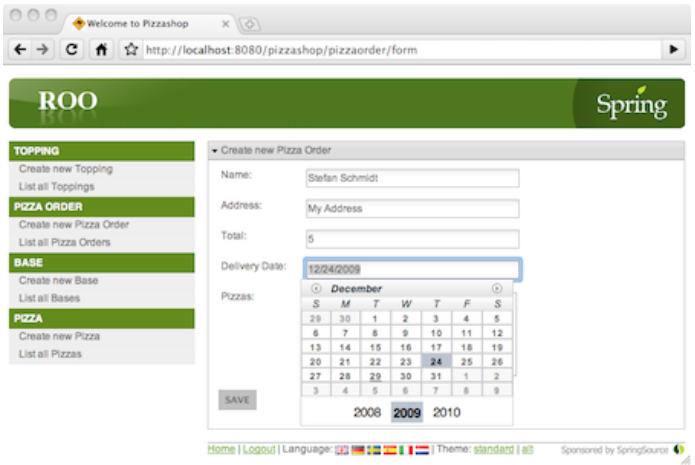
Resource	GET	PUT	POST	DELETE
Collection URI such as http://domain.com/pizzashop/topping/	List the members of the collection. For example list all the toppings available in the application.	Not used.	Create a new topping in the collection where the ID is assigned automatically by the collection.	Not used.
Member URI such as http://domain.com/pizzashop/topping/5	Retrieve the addressed topping with id=5	Update the addressed topping with id=5.	Not used.	Delete the addressed topping with id=5.
Member URI such as http://domain.com/pizzashop/topping/form	Create Form - returns an initialized, but empty topping for form binding.	Not used.	Not used.	Not used.
Member URI such as http://domain.com/pizzashop/topping/5/form	Update Form returns the topping resource which is pre-populated for form binding	Not used.	Not used.	Not used.

2.8. Step 6: Loading the Web Server

To deploy your application in a Web container during project development you have several options available:

- Deploy from your shell / command line (without the need to assemble a war archive):
  - run 'mvn tomcat:run' in the root of your project (not inside the Roo shell) to deploy to a Tomcat container
  - run 'mvn jetty:run' in the root of your project (not inside the Roo shell) to deploy to a Jetty container
- Deploy to a integrated Web container configured in STS:
  - Drag your project to the desired Web container inside the STS server view
  - Right-click your project and select 'Run As > Run on Server' to deploy to the desired Web container

After selecting your preferred deployment method you should see the Web container starting and the application should be available under the following URL http://localhost:8080/pizzashop



2.9. Securing the Application

As discussed with the Pizza Shop owner we need to control access to certain views in the Web frontend. Securing access to different views in the application is achieved by installing the Spring Security addon via the **security setup** command:

```
~.web roo> security setup
Created SPRING_CONFIG_ROOT/applicationContext-security.xml
Created SRC_MAIN_WEBAPP/WEB-INF/views/login.jspx
Updated SRC_MAIN_WEBAPP/WEB-INF/views/views.xml
Updated ROOT/pom.xml [added property 'spring-security.version' = '3.1.0.RELEASE'; added
dependencies
org.springframework.security:spring-security-core:${spring-security.version},
org.springframework.security:spring-security-config:${spring-security.version},
org.springframework.security:spring-security-web:${spring-security.version},
org.springframework.security:spring-security-taglibs:${spring-security.version}]
Updated SRC_MAIN_WEBAPP/WEB-INF/web.xml
Updated SRC_MAIN_WEBAPP/WEB-INF/spring/webmvc-config.xml
```

Note, the Roo shell will hide the **security setup** command until you have created a Web layer. As shown above, the **security setup** command manages the project `pom.xml` file. This means additional dependencies have been added to the project. To add these dependencies to the STS workspace you should run the **perform eclipse** command again followed by a project refresh (if you're using STS or m2eclipse, the "perform eclipse" command should be skipped as it will automatically detect and handle the addition of Spring Security to your project).

In order to secure the views for the `Topping`, `Base,` `` `and `` `Pizza` resources in the Pizza Shop application you need to open the `applicationContext-security.xml` file in the `src/main/resources/META-INF/spring` folder:

```
<!-- HTTP security configurations -->
<http auto-config="true" use-expressions="true">
   <form-login login-processing-url="/static/j_spring_security_check" login-page="/login"

                                          authentication-failure-url=
"/login?login_error=t"/>
   <logout logout-url="/static/j_spring_security_logout"/>
   <!-- Configure these elements to secure URIs in your application -->
   <intercept-url pattern="/pizzas/**" access="hasRole('ROLE_ADMIN')"/>
   <intercept-url pattern="/toppings/**" access="hasRole('ROLE_ADMIN')"/>
   <intercept-url pattern="/bases/**" access="hasRole('ROLE_ADMIN')"/>
   <intercept-url pattern="/resources/**" access="permitAll" />
   <intercept-url pattern="/static/**" access="permitAll" />
   <intercept-url pattern="/**" access="permitAll" />
</http>
```

As a next step you can use the Spring Security JSP tag library to restrict access to the relevant menu items in the `menu.jspx` file:

```
<div xmlns:jsp="..." xmlns:sec="http://www.springframework.org/security/tags" id="menu"
version="2.0">
   <jsp:directive.page contentType="text/html;charset=UTF-8"/>
   <jsp:output omit-xml-declaration="yes"/>
   <menu:menu id="_menu" z="nZaf43BjUg1iM0v70HJVEsXDopc=">
       <sec:authorize ifAllGranted="ROLE_ADMIN">
           <menu:category id="c_topping" z="Xm13w68rCIyzL6WIzqBtcpfiNQU=">
              <menu:item id="i_topping_new" .../>
              <menu:item id="i_topping_list" .../>
           </menu:category>
           <menu:category id="c_base" z="yTpmmNMm/hWoy3yf+aPcdUX2At8=">
              <menu:item id="i_base_new" .../>
              <menu:item id="i_base_list" .../>
           </menu:category>
           <menu:category id="c_pizza" z="mXqKC1ELexS039/pkkCrZVcSry0=">
              <menu:item id="i_pizza_new" .../>
              <menu:item id="i_pizza_list" .../>
           </menu:category>
       </sec:authorize>
       <menu:category id="c_pizzaorder" z="gBYiBODEJrzQe3q+el5ktXISc4U=">
           <menu:item id="i_pizzaorder_new" .../>
           <menu:item id="i_pizzaorder_list" .../>
       </menu:category>
    </menu:menu>
</div>
```

This leaves the pizza order view visible to the public. Obviously the delete and the update use case for the pizza order view are not desirable. The easiest way to take care of this is to adjust the `@RooWebScaffold` annotation in the `PizzaOrderController.java` source:

```
@RooWebScaffold(path = "pizzaorder",
                formBackingObject = PizzaOrder.class,
                delete=false,
                update=false)
```

This will trigger the Roo shell to remove the delete and the update method from the `PizzaOrderController` and also adjust the relevant view artifacts.

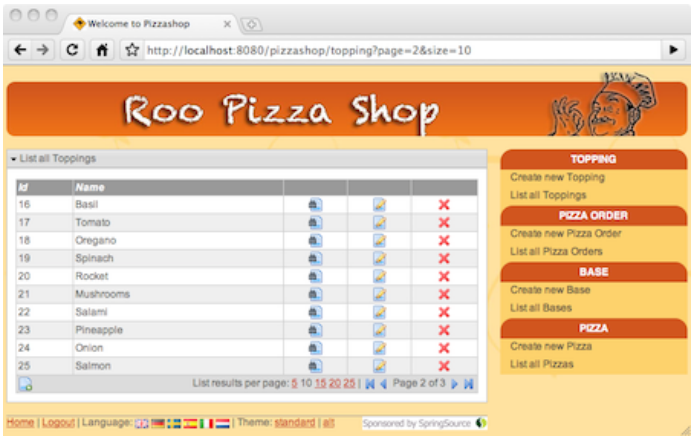
With these steps completed you can restart the application and the 'admin' user can navigate to http://localhost:8080/pizzashop/login to authenticate.

2.10. Customizing the Look & Feel of the Web UI

Roo generated Web UIs can be customized in various ways. To find your way around the installed Web-tier artifacts take a look at the following table:

Directory	Purpose
/styles	style sheets (CSS)
/images	graphics
/WEB-INF/classes/*.properties	theme configurations
/WEB-INF/config/*.xml	Web-related Spring application contexts
/WEB-INF/i18n/*.properties	internationalization message files
/WEB-INF/layouts/layout.jspx	Tiles definition for master layout
/WEB-INF/tags/*.tagx	Tag libraries (pagination, language, etc)
/WEB-INF/views/**/*	Tiles and other view artifacts
/WEB-INF/web.xml	Web application context
/WEB-INF/urlrewrite.xml	URL rewrite filter configuration

The easiest way to customize the look & feel of the Roo Web UI is to change CSS and image resources to suit your needs. The following look & feel was created for the specific purpose of the Pizza Shop application:



Spring Roo also configures theming support offered by Spring framework so you can leverage this feature with ease.

To achieve a higher level of customization you can change the default Tiles template (WEB-INF/layouts/default.jspx) and adjust the JSP pages (WEB-INF/views/*.jspx). WIth release 1.1 of Spring Roo jspx artifacts can now be adjusted by the user while Roo can still make adjustments as needed if domain layer changes are detected. See the JSP Views section for details.

Furthermore the Spring Roo 1.1 release introduced a set of JSP tags which not only reduce the scaffolded jspx files by 90% but also offer the most flexible point for view customization. Roo will install these tags into the user project where they can be accessed and customized to meet specific requirements of the project. For example it would be fairly easy to remove the integrated Spring JS / Dojo artifacts and replace them with your JS framework of choice. To make these changes available for installation in other projects you can create a simple add-on which replaces the default tags installed by Roo with your customized tags.

2.11. Selenium Tests

Roo offers a core addon which can generate Selenium test scripts for you. You can create the Selenium scripts by using the **selenium test** command. Tests are generated for each controller and are integrated in a test suite:

```
~.web roo> selenium test --controller ~.web.ToppingController
~.web roo> selenium test --controller ~.web.BaseController
~.web roo> selenium test --controller ~.web.PizzaController
~.web roo> selenium test --controller ~.web.PizzaOrderController
```

The generated tests are located in the `src/main/webapp/selenium` folder and can be run via the following maven command (executed from command line, not the Roo shell):

```
pizza> mvn selenium:selenese
```

Running the maven selenium addon will start a new instance of the FireFox browser and run tests against the Pizza Shop Web UI by using Roo generated seed data.

Please note that the maven selenium plugin configured in the project `pom.xml` assumes that the FireFox Web browser is already installed in your environment. Running the maven selenium plugin also assumes that your application is already started as discussed in step 6. Finally, there are limitations with regards to locales used by the application. Please refer to the known issues section for details.

2.12. Backups and Deployment

One other very useful command is the **backup** command. Issuing this command will create you a backup of the current workspace with all sources, log files and the script log file (excluding the target directory):

```
~.web roo> backup
Created ROOT/pizzashop_2012-02-14_18:10:19.zip
Backup completed in 35 ms
~.web roo>
```

Finally, you may wish to deploy your application to a production Web container. For this you can easily create a war archive by taking advantage of the **perform package** command:

```
~.web roo> perform package
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building pizzashop
[INFO]    task-segment: [package]
[INFO] ------------------------------------------------------------------------
...
[INFO] [war:war {execution: default-war}]
[INFO] Exploding webapp...
[INFO] Assembling webapp pizzashop in /Users/stewarta/projects/roo-
test/pizzashop/target/pizzashop-0.1.0-SNAPSHOT
[INFO] Copy webapp webResources to /Users/stewarta/projects/roo-
test/pizzashop/target/pizzashop-0.1.0-SNAPSHOT
[INFO] Generating war /Users/stewarta/projects/roo-test/pizza/target/pizzashop-0.1.0-
SNAPSHOT.war
[INFO] Building war: /Users/stewarta/projects/roo-test/pizza/target/pizzashop-0.1.0-
SNAPSHOT.war
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 5.881s
[INFO] Finished at: Tue Feb 14 18:07:54 EST 2012
[INFO] Final Memory: 8M/81M
[INFO] ------------------------------------------------------------------------
~.web roo>
```

This command produces your war file which can then be easily copied into your production Web container.

2.13. Where To Next

Congratuations! You've now completed the Roo Pizza Shop tutorial. You're now in a good position to try Roo for your own projects. While reading the next few chapters of this reference guide will help you understand more about how to use Roo, we suggest the following specific sections if you'd like to know more about Roo:

- Roo Addon Suites
- Add-On management using OSGi Repositories
- Spring Roo Marketplace

Chapter 3. Application Architecture

In this chapter we'll introduce the architecture of Roo-created projects. In later chapters we'll cover the architecture of Roo itself.

This chapter focuses on web applications created by Roo, as opposed to add-on projects.

3.1. Architectural Overview

Spring Roo focuses on the development of enterprise applications written in Java. In the current version of Roo these applications typically will have a relational database backend, Java Persistence API (JPA) persistence approach, Spring Framework dependency injection and transactional management, JUnit tests, a Maven build configuration and usually a Spring MVC-based front-end that uses JSP for its views. As such a Roo-based application is like most modern Java-based enterprise applications.

While most people will be focusing on developing these Spring MVC-based web applications, it's important to recognise that Roo does not impose any restrictions on the sort of Java applications that can be built with it. Even with Roo it was easy to build any type of self-contained application. Some examples of the types of requirements you can easily address with the current version of Roo include (but are not limited to):

- Listening for messages on a JMS queue and sending replies over JMS or SMTP (Roo can easily set up JMS message producers, consumers and SMTP)
- Writing a services layer (perhaps annotated with Spring's @Service stereotype annotation) and exposing it using a remoting protocol to a rich client (Spring's remoting services will help here)
- Executing a series of predefined actions against the database, perhaps in conjunction with Spring's new @Scheduled or @Async timer annotations
- Experimentation with the latest Spring and AspectJ features with minimal time investment

One of the major differences between Roo and traditional, hand-written applications is we don't add layers of abstraction unnecessarily. Most traditional Java enterprise applications will have a DAO layer, services layer, domain layer and controller layer. In a typical Roo application you'll only use an entity layer (which is similar to a domain layer) and a web layer. As indicated by the list above, a services layer might be added if your application requires it, although a DAO layer is extremely rarely added. We'll look at some of these layering conventions (and the rationale for them) as we go through the rest of this chapter.

3.2. Critical Technologies

Two technologies are very important in all Roo projects, those being AspectJ and Spring. We'll have a look at how Roo-based applications use these technologies in this section.

3.2.1. AspectJ

AspectJ is a powerful and mature aspect oriented programming (AOP) framework that underpins many large-scale systems. Spring Framework has offered extensive support for AspectJ since 2004, with Spring 2.0 adopting AspectJ's pointcut definition language even for expressing Spring AOP pointcuts. Many of the official Spring projects offer support for AspectJ or are themselves heavily dependent on it, with several examples including Spring Security (formerly Acegi Security System for Spring), Spring Insight, SpringSource tc Server, SpringSource dm Server, Spring Enterprise and Spring Roo.

While AspectJ is most commonly known for its aspect oriented programming (AOP) features such as applying advice at defined pointcuts, Roo projects use AspectJ's powerful inter-type declaration (ITD) features. This is where the real magic of Roo comes from, as it allows us to code generate members (artifacts like methods, fields etc) in a different compilation unit (i.e. source file) from the normal .java code you'd write as a developer. Because the generated code is in a separate file, we can maintain that file's lifecycle and contents completely independently of whatever you are doing to the .java files. Your .java files do not need to do anything unnatural like reference the generated ITD file and the whole process is completely transparent.

Let's have a look at how ITDs work. In a new directory, type the following commands and note the console output:

```
roo> <strong>project setup --topLevelPackage com.aspectj.rocks</strong>
roo> <strong>jpa setup --database HYPERSONIC_IN_MEMORY --provider HIBERNATE</strong>
roo> <strong>entity jpa --class ~.Hello</strong>
Created SRC_MAIN_JAVA/com/aspectj/rocks
Created SRC_MAIN_JAVA/com/aspectj/rocks/Hello.java
Created SRC_MAIN_JAVA/com/aspectj/rocks/Hello_Roo_JpaActiveRecord.aj
Created SRC_MAIN_JAVA/com/aspectj/rocks/Hello_Roo_JpaEntity.aj
Created SRC_MAIN_JAVA/com/aspectj/rocks/Hello_Roo_ToString.aj
Created SRC_MAIN_JAVA/com/aspectj/rocks/Hello_Roo_Configurable.aj
roo> field string --fieldName comment
Managed SRC_MAIN_JAVA/com/aspectj/rocks/Hello.java
Managed SRC_MAIN_JAVA/com/aspectj/rocks/Hello_Roo_JavaBean.aj
Managed SRC_MAIN_JAVA/com/aspectj/rocks/Hello_Roo_ToString.aj
```

Notice how there is a standard `Hello.java` file, as well as a series of `Hello_Roo_*.aj` files. Any file ending in *_Roo_*.aj is an AspectJ ITD and will be managed by Roo. You should not edit these files directly, as Roo will automatically maintain them (this includes even deleting files that aren't required, as we'll see shortly).

The `Hello.java` is just a normal Java file. It looks like this:

```
package com.aspectj.rocks;

import org.springframework.roo.addon.javabean.RooJavaBean;
import org.springframework.roo.addon.tostring.RooToString;
import org.springframework.roo.addon.entity.RooJpaActiveRecord;

@RooJavaBean
@RooToString
@RooJpaActiveRecord
public class Hello {

    private String comment;
}
```

As shown, there's very little in the `.java` file. There are some annotations, plus of course the field we added. Note that Roo annotations are always source-level retention, meaning they're not compiled into your `.class` file. Also, as per our usability goals you'll note that Roo annotations also always start with `@Roo*` to help you find them with code assist.

By this stage you're probably wondering what the ITD files look like. Let's have a look at one of them, `Hello_Roo_ToString.aj`:

```
package com.aspectj.rocks;

import org.apache.commons.lang3.builder.ReflectionToStringBuilder;
import org.apache.commons.lang3.builder.ToStringStyle;

privileged aspect Hello_Roo_ToString {

    public String Hello.toString() {
          return ReflectionToStringBuilder.toString(this, ToStringStyle
.SHORT_PREFIX_STYLE);
    }

}
```

Notice how the ITD is very similar to Java code. The main differences are that it is declared with "privileged aspect", plus each member identifies the target type (in this case it is "Hello.toString", which means add the "toString" method to the "Hello" type). The compiler will automatically recognize these ITD files and cause the correct members to be compiled into `Hello.class`. We can see that quite easily by using Java's `javap` command. All we need to do is run the compiler and view the resulting class. From the same directory as you created the project in, enter the following commands and observe the final output:

```
$ <strong>mvn compile</strong>
$ <strong>javap -classpath target/classes/.:target/test-classes/.
com.aspectj.rocks.Hello</strong>
Compiled from "Hello.java"
public class com.aspectj.rocks.Hello extends java.lang.Object implements
org.springframework.beans.factory.aspectj.ConfigurableObject{
    transient javax.persistence.EntityManager entityManager;
    public com.aspectj.rocks.Hello();
    public static java.lang.String ajc$get$comment(com.aspectj.rocks.Hello);
    public static void ajc$set$comment(com.aspectj.rocks.Hello, java.lang.String);
    public static java.lang.Long ajc$get$id(com.aspectj.rocks.Hello);
    public static void ajc$set$id(com.aspectj.rocks.Hello, java.lang.Long);
    public static java.lang.Integer ajc$get$version(com.aspectj.rocks.Hello);
    public static void ajc$set$version(com.aspectj.rocks.Hello, java.lang.Integer);
    static {};
    public static long countHelloes();
    public static final javax.persistence.EntityManager entityManager();
    public static java.util.List findAllHelloes();
    public static com.aspectj.rocks.Hello findHello(java.lang.Long);
    public static java.util.List findHelloEntries(int, int);
    public void flush();
    public java.lang.String getComment();
    public java.lang.Long getId();
    public java.lang.Integer getVersion();
    public com.aspectj.rocks.Hello merge();
    public void persist();
    public void remove();
    public void setComment(java.lang.String);
    public void setId(java.lang.Long);
    public void setVersion(java.lang.Integer);
    public java.lang.String toString();
}
```

While the `javap` output might look a little daunting at first, it represents all the members that Roo has added (via AspectJ ITDs) to the original `Hello.java` file. Notice there isn't just the `toString` method we saw in the earlier ITD, but we've also made the `Hello` class implement Spring's `ConfigurableObject` interface, provided access to a JPA `EntityManager`, included a range of convenient persistence methods plus even getters and setters. All of these useful features are automatically maintained in a round-trip compatible manner via the ITDs.

A careful reader might be wondering about the long field names seen for introduced fields. You can see that these field names start with "ajc$" in the output above. The reason for this is to avoid name collisions with fields you might have in the `.java` file. The good news is that you won't ever need to deal with this unless you're trying to do something clever with reflection. It's just something to be aware of for introduced fields in particular. Note that the names of methods and constructors are never modified.

Naturally as a normal Roo user you won't need to worry about the internals of ITD source code and the resulting `.class` files. Roo automatically manages all ITDs for you and you never need deal with them directly. It's just nice to know how it all works under the hood (Roo doesn't believe in magic!). The benefit of this ITD approach is how easily and gracefully Roo can handle code generation for you.

To see this in action, go and edit the `Hello.java` in your favourite text editor with Roo running. Do something simple like add a new field. You'll notice the `Hello_Roo_ToString.aj` and `Hello_Roo_JavaBean.aj` files are instantly and automatically updated by Roo to include your new field. Now go and write your own `toString` method in the `.java` file. Notice Roo deletes the `Hello_Roo_ToString.aj` file, as it detects your `toString` method should take priority over a generated `toString` method. But let's say you want a generated `toString` as well, so change the `Hello.java`'s `@RooToString` annotation to read `@RooToString(toStringMethod="generatedToString")`. Now you'll notice the `Hello_Roo_ToString.aj` file is immediately re-created, but this time it introduces a `generatedToString` method instead of the original `toString`. If you comment out both fields in `Hello.java` you'll also see that Roo deletes both ITDs. You can also see the same effect by quitting the Roo shell, making any changes you like, then restarting the Roo shell. Upon restart Roo will automatically perform a scan and discover if it needs to make any changes.

Despite the admittedly impressive nature of ITDs, AspectJ is also pretty good at aspect oriented programming features like pointcuts and advice! To this end Roo applications also use AspectJ for all other AOP requirements. It is AspectJ that provides the AOP so that classes are dependency injected with singletons when instantiated and transactional services are called as part of method invocations. All Roo applications are preconfigured to use the Spring Aspects project, which ships as part of Spring Framework and represents a comprehensive "aspect library" for AspectJ.

3.2.2. Spring

Spring Roo applications all use Spring. By "Spring" we not only mean Spring Framework, but also the other Spring projects like Spring Security. Of course, only Spring Framework is installed into a user project by default and there are fine-grained commands provided to install each additional Spring project beyond Spring Framework.

All Roo applications use Spring Aspects, which was mentioned in the AspectJ section and ensures Spring Framework's `@Configurable` dependency injection and transactional advice is applied. Furthermore, Roo applications use Spring's annotation-driven component scanning by default and also rely on Spring Framework for instantiation and dependency injection of features such as JPA providers and access to database connection pools. Many of the optional features that can be used in Roo applications (like JMS and SMTP messaging) are also built upon the corresponding Spring Framework dependency injection support and portable service abstractions.

Those Roo applications that include a web controller will also receive Spring Framework 3's MVC features such as its conversion API, web content negotiation view resolution and REST support. It is possible (and indeed encouraged) to write your own web Spring MVC controllers in Roo applications, and you are also free to use alternate page rendering technologies if you wish (i.e. not just JSP).

Generally speaking Roo will not modify any Spring-related configuration or setting file (e.g. properties) unless specifically requested via a shell command. Roo also ensures that whenever it creates, modifies or deletes a file it explicitly tells you about this via a shell message. What this means is you can safely edit your Spring application context files at any time and without telling Roo. This is very useful if the default configuration offered by Roo is unsuitable for your particular application's needs.

Because Spring projects are so extensively documented, and Roo just uses Spring features in the normal manner, we'll refrain from duplicating Spring's documentation in this section. Instead please refer to the excellent Spring documentation for guidance, which can be found in the downloadable distribution files and also on the Spring web site.

3.3. Entity Layer

When people use Roo, they will typically start a new project using the steps detailed in the Beginning With Roo: The Tutorial chapter. That is, they'll start by creating the project, installing some sort of persistence system, and then beginning to create entities and add fields to them. As such, entities and fields represent the first point in a Roo project that you will be expressing your problem domain.

The role of an entity in your Roo-based application is to model the persistent "domain layer" of your system. As such, a domain object is specific to your problem domain but an entity is a special form of a domain object that is stored in the database. By default a single entity will map to a single table in your database, and a single field within your entity class will map to a single column within the corresponding table. However, like most things in Roo this is easily customised using the relevant standard (in this case, JPA annotations). Indeed most of the common customisation options (like specifying a custom column or table name etc) can be expressed directly in the relevant Roo command, freeing you from even needing to know which annotation(s) should be used.

Let's consider a simple entity that has been created using the entity jpa command and following it with a single field command:

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

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

@RooJavaBean
@RooToString
@RooJpaActiveRecord
public class Choice {

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

    @Size(max = 80)
    private String description;
}
```

The above entity is simply a JPA entity that contains two fields. The two fields are annotated with JavaBean Validation API (JSR 303) annotations, which are useful if your JPA provider supports this standard (as is the case if you nominate Hibernate as your JPA provider) or you are using a Roo-scaffolded web application front end (in which case Roo will use Spring Framework 3's JSR 303 support). Of course you do not need to use the JavaBean Validation API annotations at all, but if you would like to use them the relevant Roo field commands provide tab-completion compatible options for each. The first time you use one of these Roo field commands, Roo will add required JavaBean Validation API libraries to your project (i.e. these libraries will not be in your project until you decide to first use JavaBean Validation).

What's interesting about the above entity is what you can actually do with it. There are a series of methods automatically added into the `Choice.class` courtesy of Roo code-generated and maintained AspectJ ITDs. These include static methods for retrieving instances of Choice, JPA facade methods for persisting, removing, merging and flushing the entity, plus accessors and mutators for both the identifier and version properties. You can fine-tune these settings by modifying attributes on the `@RooJpaActiveRecord` annotation. You can also have Roo remove these services by simply removing the `@RooJpaActiveRecord` annotation from the class, in which case you'll be left with a normal JPA @Entity that you'll need to manage by hand (e.g. provide your own persistence methods, identifier, version etc).

The `@RooJavaBean` annotation causes an accessor and mutator (getter and setter) to automatically be generated for each field in the class. These accessors and mutators are automatically maintained in an AspectJ ITD by Roo. If you write your own accessor or mutator in the normal .java file, Roo will automatically remove the corresponding generated method from the ITD. You can also remove the

`@RooJavaBean` annotation if you don't want any generated accessors or mutators (although those related to the version and identifier fields will remain, as they are associated with `@RooJpaActiveRecord` instead of `@RooJavaBean`).

Finally, the `@RooToString` annotation causes Roo to create and maintain a `public String toString()` method in a separate ITD. This method currently is used by any scaffolded web controllers if they need to display a related entity. The generated method takes care to avoid circular references that are commonly seen in bidirectional relationships involving collections. The method also formats Java `Calendar` objects in an attractive manner. As always, you can write your own `toString()` method by hand and Roo will automatically remove its generated `toString()` method, even if you still have the `@RooToString` annotation present. You can of course also remove the `@RooToString` annotation if you no longer wish to have a generated `toString()` method.

Before leaving this discussion on entities, it's worth mentioning that you are free to create your own entity `.java` classes by hand. You do not need to use the Roo shell commands to create entities or maintain their fields - just use any IDE. Also, you are free to use the `@RooToString` or `@RooJavaBean` (or both) annotations on any class you like. This is especially useful if you have a number of domain objects that are not persisted and are therefore not entities. Roo can still help you with those objects.

3.4. Web Layer

Spring Roo can optionally provide a scaffolded Spring MVC web layer. The scaffolded MVC web layer features are explored in some depth in the Beginning With Roo: The Tutorial chapter, including how to customise the appearance. From an architectural perspective, the scaffolded layer includes a number of URL rewriting rules to ensure requests can be made in accordance with REST conventions. Roo's scaffolding model also includes Apache Tiles, Spring JavaScript, plus ensures easy setup of Spring Security with a single command.

Scaffolded web controllers always delegate directly to methods provided on an `@RooJpaActiveRecord` class. For maximum compatibility with scaffolded controllers, it is recommended to observe the default identifier and version conventions provided by `@RooJpaActiveRecord` implementations. If you write a web controller by hand (perhaps with the assistance of the web mvc controller command), it is recommended you also use the methods directly exposed on entities. Most Roo applications will place their business logic between the entities and web controllers, with only occasional use of services layers. Please refer to the services layer section for a more complete treatment of when you'd use a services layer.

3.5. Services Layer

As discussed at the start of this chapter, web applications are the most common type of application created with Roo. A web application will rarely *require* a services layer, as most logic can be placed in the web controller handle methods and the remainder in entity methods. Still, a services layer makes sense in specific scenarios such as:

- There is business logic that spans multiple entities and that logic does not naturally belong in a

specific entity

- You need to invoke business logic outside the scope of a natural web request (e.g. a timer task)
- Remote client access is required and it is therefore more convenient to simply expose the methods via a remoting protocol
- An architectural policy requires the use of a services layer
- A higher level of cohesion is sought in the web layer, with the web layer solely responsible for HTTP-related management and the services layer solely responsible for business logic
- A greater level of testing is desired, which is generally easier to mock than simulating web requests
- it is preferred to place transactional boundaries and security authorization metadata on the services layer (as opposed to a web controller)

As shown, there are a large number of reasons why services layers remain valuable. However, Roo does not code generate services layers because they are not strictly essential to building a normal web application and Roo achieves separation of concern via its AspectJ ITD-based architecture.

If you would like to use a services layer, Roo offers automatic service layer integration for your application. Please refer to the service layer section in the application layering chapter for further details.

3.6. Repository Layer

Repository layer is not generated automatically using current version of Spring Roo, but we recommed to include a Repository Layer on generated projects.

Roo offers support for different repository layers as of release 1.2.0. Please refer to the application layering chapter for details.

In future versions of Spring Roo, this layer will be generated automatically.

3.7. Maven

3.7.1. Packaging

Roo supports a number of Maven packaging types out of the box, such as `jar`, `war`, `pom`, and `bundle`. These are provided via Roo's `PackagingProvider` interface. If you wish to customise the POMs or other artifacts that Roo generates for a given packaging type when creating a project or module, either for one of the above packaging types or a completely different one, you can implement your own `PackagingProvider` that creates exactly the files you want with the contents you want. The procedure for doing this is as follows:

In a new directory, start Roo and run "addon create simple" to create a simple addon.

- Delete:
  - the four .java files created in `src/main/java`
  - the two .tagx files created in `src/main/resources`
- Create your custom packaging class (e.g. `MyPackaging.java`) in your preferred package.
- Pick a unique ID for the Roo shell to use when referring to your PackagingProvider (e.g. "custom-jar"). Do not use any of the core Maven packaging type names, as these are reserved for use by Roo.
- Make your packaging class implement the `o.s.r.project.packaging.PackagingProvider` interface, either by:
  - Implementing `PackagingProvider` directly, with full control over (but no assistance with) artifact generation, or
  - Extending `o.s.r.project.packaging.AbstractPackagingProvider` to have Roo create the POM from a template you specify, with various substitutions made automatically (e.g. groupId and artifactId). This approach requires you to:
    - Create your custom POM template in `src/main/resources` plus whatever package you chose above.
    - Create a public no-arg constructor that calls the `AbstractPackagingProvider` constructor with the following arguments:
      - The unique ID of your custom packaging type (see above).
      - The Maven name of your packaging type (typically jar/war/ear/etc, but could be something else if you've extended Maven to support custom packaging types).
      - The path to your POM template relative to your concrete `PackagingProvider` (e.g. "my-pom-template.xml" if it's in the same package). Note that this POM can contain as much or as little content as you like, with the following caveats:
        - It must have the standard Maven "project" root element with all the usual namespace details.
        - If you extend `AbstractPackagingProvider`, that class will ensure that the POM's coordinates can be resolved either from a "parent" element or from explicit "groupId", "artifactId", and "version" elements.
- Add the Felix annotations @Component and @Service to your concrete PackagingProvider, so that it's detected by Roo's `PackagingProviderRegistry`.
- Build and install the addon in the usual way, i.e.:

- Run “mvn install” in the addon directory to create the OSGi bundle.
- Change to the directory of the project that will be using the custom packaging provider.
- Run `"osgi start --url file://path/to/addon/project/target/com.example.foo-0.1.0.BUILD-SNAPSHOT.jar"`
- Run `"osgi scr list"`; your custom PackagingProvider component should appear somewhere in the list.

- Whenever you run the "project" or "module create" commands, your custom PackagingProvider’s ID should appear in the list of possible completions for the "--packaging" option

3.7.2. Multi-Module Support

Since version 1.2.0, Roo supports multi-module Maven projects, i.e. those containing multiple projects in a nested directory structure, each with their own POM. The non-leaf POMs have "pom" packaging and the leaf POMs usually have an artifact creation packaging (jar, war, etc). If you’re not familiar with multi-module projects and want to see how they’re structured, there’s an embedded `multimodule.roo` script that generates a simple multi-module project; used as follows:

- At your operating system prompt, type “roo script multimodule.roo”.
- Change into the "ui/mvc"" directory.
- Run “mvn tomcat:run” or “mvn jetty:run”.
- Point your browser to `http://localhost:8080/mvc`.

The rest of this section assumes that you are familiar with multi-module projects, in particular the difference between POM inheritance (one POM has another as its parent) and project nesting (one project is in a sub-directory of another, i.e. is a module of that parent project).

3.7.2.1. Features

Roo’s multi-module support has the following features (a formal list of Roo’s Maven-related commands appears in Appendix C):

- Roo now has the concept of a module, which in practice means a directory tree whose root contains a Maven POM. A project consists of zero or more modules. When you run Roo from the operating system prompt, you do so from the directory of the root module.

  Once any modules exist, one of them always has the "focus", in other words will be used as the context for any shell commands that interact with the user project (as opposed to housekeeping commands such as “osgi ps”). For example, running the “web flow” command will add Spring Web Flow support to the currently focused module.

- The “module focus” command, available once the project contains more than one module, changes

the currently focused module. Tab completion is available, with the module name "~" signifying the root module.

- The “module create” command creates a new module as a sub-directory of the currently focused module. The latter module’s POM will be updated to ensure it has "pom" packaging, allowing the Maven reactor to properly recurse the module tree at build time. Note that the newly created POM will by default *not* inherit from the parent module’s POM. If the new module’s POM should have a parent, specify it via the “module create” command’s optional “parent” parameter. The parent POM need not be located within the user project. A typical use case is that a development team might have a standard base POM from which all their projects inherit, or a standard web POM from which all their web modules inherit. As with the “project” command, the new module’s Maven packaging can be specified via the optional “packaging” parameter. Custom packaging behaviour is supported, as described above.

3.7.2.2. Limitations

Roo’s multi-module support has the following limitations:

- Limited automatic creation of dependencies between modules. If your project needs any inter-module dependencies beyond those added by Roo, simply create them using the "dependency add" command.

- No command for removing a module; this is in line with the absence of commands for removing other project artifacts such as classes, enums, JSPs, and POMs. In any event, it’s simple enough to do manually; just delete the directory, delete the relevant "`<module>`" element from the parent module’s POM, and delete the module as a dependency from any other modules' POMs.

- One area where there’s considerable scope for improvement is in the management of dependencies in general. In an ideal Maven project, dependency information in the form of both "`dependencyManagement`" entries and live "`dependency`" elements themselves would be pushed as far up the POM inheritance hierarchy as possible, in order to minimise duplication and reduce the incidence of version conflicts. As it stands, Roo adds and removes dependencies to and from the currently focused module in response to shell commands, regardless of what dependencies are in effect for other modules in the project.

- Likewise, plugin management is currently quite basic. Roo adds/removes plugins to the POM of the currently focused module with no attempt to rationalise them in concert with the POMs of other modules (for example, two Spring MVC modules will independently have the Jetty plugin declared in their own POMs rather than having this plugin declared in the lowest common ancestor POM). As with dependencies (see above), this is an area in which Roo could conceivably take some of the load off developers.

- There’s no Roo command for changing a module’s packaging between two arbitrary values, as this could require too many other changes to the user’s project. However, Roo does change a module’s packaging in two specific circumstances:

  - Adding a module to the currently focused module changes the latter’s packaging to "pom", as

described above under the "`module create`" command.

    - Adding web support to a module changes its packaging to "war".

- Roo does not create any parent-child relationships between different modules' Spring application contexts; the user can always create these relationships manually, and Roo will not remove them.

II. Roo Add-Ons

This part of the reference guide provides a detailed reference to the major Roo base add-ons and how they work. This part goes into more detail than the tutorial chapter and offers a "bigger picture" discussion than the command reference appendix.

Chapter 4. Add-Ons Overview

When you download the Spring Roo distribution ZIP, there are actually two major logical components in use. The first of these is the "*Roo core*", which provides an environment in which to host add-ons and provide services to them. The other component is what we call "*base add-ons*". A base add-on is different from a third party add-on only in that it is included in the Roo distribution by default and does not require you to separately install it. In addition, you cannot remove a base add-on using normal Roo commands.

Base add-ons always adopt the package name prefix `org.springframework.roo.addon`. We also have a part of Roo known as "Roo core". This relates to the core modules, and these always have package names that start with `org.springframework.roo` (but excluding those with “addon” as the next package name segment, as in that case they’d be a "base add-on"). Roo core provides very few commands, and whatever commands it provides are generally internal infrastructure-related features (like "project scan status" or "metadata for id") or sometimes aggregate the features provided by several individual base add-ons (e.g. "`entity jpa --testAutomatically`").

Add-ons that do not ship with Spring Roo but are nevertheless about to be used with it could be located on link:#roo-marketplace and could be installed using OSGi Repositories.

Of course as a user of Roo you do not need to be aware of whether a particular component is part of Roo core, a base add-on or an installable add-on. It’s just useful for us to formally define these commonly-used terms and explain the impact on whether you need to install or uninstall a component or not.

The individual base add-ons provided by Roo provide capabilities in the following key functional areas:

- Project management (like project creation, dependency management, "perform" commands)
- General type management (like creation of types, toString method, JavaBean methods)
- Persistence (like JPA setup, entities)
- Field management (like field creation with JPA compliance)
- Database introspection and reverse engineering
- Dynamic finders (creation of finders without needing to write the JPA-QL for them)
- JUnit testing (with integration and mock testing)
- Spring MVC (including URL rewriting, JSP services, controller management)
- Spring Security
- Selenium testing

- Java Message Service (JMS)
- Simple Mail Transfer Service (SMTP)
- Log4J configuration
- Add-On Management

We have added dedicated chapters for many of these functional areas in this, Part II of our documentation.

Chapter 5. Persistence Add-On

The persistence add-on provides a convenient way to create Java Persistence API (JPA v2) compliant entities. There are different commands available to configure JPA, create new JPA-compliant entities, and add fields to these entities. In the following a summary of the features offered by the Spring Roo persistence add-on:

5.1. JPA setup command

The jpa setup command provides the following options and attributes:

Database Options:

- HSQL (in memory)
- HSQL (persistent)
- H2 (in memory)
- MySQL
- Postgres
- MS SQL Server
- Sybase
- Oracle *
- DB2 *
- DB2/400
- Apache Derby (Java DB)
- Firebird

* The JDBC driver dependencies for these databases are not available in public Maven repositories. As such, Roo configures a default dependency in your project `pom.xml`. You need to adjust it according to your specific version of your database driver available in your private Maven repository.

Some useful hints to get started with Oracle Express (Oracle XE): After installing Oracle XE you need to find the JDBC driver under `${oracle-xe}/app/oracle/product/10.2.0/server/jdbc/lib` and run the command:

```
mvn install:install-file -Dfile=ojdbc14_g.jar -DgroupId=com.oracle -DartifactId=ojdbc14
-Dversion=10.2.0.2 -Dpackaging=jar -DgeneratePom=true
```

Also, if you dont want Jetty (or Tomcat) to be conflicting with oracle-xe web-server, you should use the following command: `mvn jetty:run -Djetty.port=8090`.

ORM Provider Options:

- EclipseLink
- Hibernate
- OpenJPA
- DataNucleus 3.0

In addition, the jpa setup command accepts optional *databaseName*, *userName* and *password* attributes for your convenience. However, it's not necessary to use this command. You can easily edit these details in the `database.properties` file at any time. Finally, you can also specify a pre-configured JNDI datasource via the *jndiDataSource* attribute.

The jpa setup command can be re-run at any time. This means you can change your ORM provider or database when you plan to move your application between your development setup (e.g. Hibernate with HSQLDB) to your production setup (e.g. EclipseLink with DB2). Of course this is a convenience only. You'll naturally experience fewer deployment issues if you use the same platform for both development and production.

Running the jpa setup command in the Roo shell takes care of configuring several aspects in your project:

1. JPA dependencies are registered in the project `pom.xml` Maven configuration. It includes the JPA API, ORM provider (and its dependencies), DB driver, Spring ORM, Spring JDBC, Commons DBCP, and Commons Pool
2. Persistence XML configuration with a persistence-unit preconfigured based on your choice of ORM provider and Database. Here is an example for the EclipseLink ORM provider and HSQL database:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="2.0"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
                                 http://java.sun.com/xml/ns/persistence/persistence_2_0
.xsd">

    <persistence-unit name="persistenceUnit" transaction-type="RESOURCE_LOCAL">
        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
        <properties>
            <property name="eclipselink.target-database"
                      value="org.eclipse.persistence.platform.database.HSQLPlatform"/>

            <!--value='drop-and-create-tables' to build a new database on each run;
                value='create-tables' creates new tables if needed;
                value='none' makes no changes to the database-->
            <property name="eclipselink.ddl-generation" value="drop-and-create-tables
"/>

            <property name="eclipselink.ddl-generation.output-mode" value="database"/>

            <property name="eclipselink.weaving" value="static"/>
        </properties>
    </persistence-unit>
</persistence>
```

By default the persistence unit is configured to build a new database on each application restart. This helps to avoid data inconsistencies during application development when the domain model is not yet finalized (new fields added to an entity will yield new table columns). If you feel that your domain model is stable you can manually switch to a mode which allows data persistence across application restarts in the persistence.xml file. This is documented in the comment above the relevant property. Each ORM provider uses different property names and values to achieve this.

3. A database properties file (`src/main/resources/META-INF/spring/database.properties`) which contains user name, password, JDBC driver name and connection URL details:

```
database.url=jdbc\:hsqldb\:mem\:foo
database.username=sa
database.password=
database.driverClassName=org.hsqldb.jdbcDriver
```

This file can be edited manually, or you can use the properties set command, or by using the *databaseName*, *userName* and *password* attributes of the jpa setup command. You can edit the properties file or use any of these commands at any time.

4. A DataSource definition and a transaction manager are added to the Spring application context:

```
[...]
<bean class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close" id=
"dataSource">
    <property name="driverClassName" value="${database.driverClassName}"/>
    <property name="url" value="${database.url}"/>
    <property name="username" value="${database.username}"/>
    <property name="password" value="${database.password}"/>
</bean>

<bean class="org.springframework.orm.jpa.JpaTransactionManager" id="transactionManager
">
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
</bean>

<tx:annotation-driven mode="aspectj" transaction-manager="transactionManager"/>

<bean class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
      id="entityManagerFactory">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

5.2. Entity JPA command

Using the link:#command-index[entity jpa] command you can create simple Java beans which are annotated with JPA annotations. There are several optional attributes which can be used as part of this command but in its simplest form it will generate the following artifacts:

```
roo> entity jpa --class ~.Person
Created SRC_MAIN_JAVA/com/foo
Created SRC_MAIN_JAVA/com/foo/Person.java
Created SRC_MAIN_JAVA/com/foo/Person_Roo_JavaBean.aj
Created SRC_MAIN_JAVA/com/foo/Person_Roo_Jpa_Entity.aj
Created SRC_MAIN_JAVA/com/foo/Person_Roo_Jpa_ActiveRecord.aj
Created SRC_MAIN_JAVA/com/foo/Person_Roo_ToString.aj
Created SRC_MAIN_JAVA/com/foo/Person_Roo_Configurable.aj
~.Person roo>
```

As you can see from the Roo shell messages there are 6 files generated (also, note that the context has changed to the Person type in the Roo shell):

1. Person.java:

```
@RooJavaBean
@RooToString
@RooJpaActiveRecord
public class Person {
}
```

You will notice that by default, the Person type does not contain any fields (these will be added with the field commands or manually in the type) or methods.

2. Person_Roo_JavaBean.aj (this will only be generated when fields are added to the Person type)

   The first annotation added by the entity jpa command is the @RooJavaBean annotation. This annotation will automatically add public accessors and mutators via an ITD for each field added to the Person type. This annotation (like all Roo annotations) has source retention (so it will not be present in the generated byte code).

3. Person_Roo_ToString.aj

   The second annotation added to the Person type is the @RooToString annotation. This annotation will generate a toString method for the Person type via an ITD. The *toString*() method will contain a concatenated representation of all field names and their values using the commons-lang RefectionToStringBuilder by default. If you want to provide your own toString() method alongside the Roo generated *toString*() method you can declare the *toStringMethod* attribute in the @RooToString annotation. This attribute allows you to change the default method name of the Roo-managed *toString*() (default name) method, thereby allowing your custom toString() method alongside the Roo-managed method.

4. Person_Roo_Configurable.aj

   This ITD is automatically created and does not require the @RooConfigurable annotation to be introduced into the Person.java type. It takes care of marking the Person type with Spring's @Configurable annotation. This annotation allows you to inject any types from the Spring bean factory into the Person type. The injection of the JPA entity manager (which is defined as a bean in the application context) is possible due to the presence of the @Configurable annotation.

5. Person_Roo_Jpa_Entity.aj

   The forth annotation is the @RooJpaActiveRecord annotation. This annotation triggers the creation of two ITDs: the Person_Roo_Jpa_Entity.aj ITD and the Person_Roo_Jpa_ActiveRecord.aj ITD. Note that If you do not want ActiveRecord-style methods in your domain object you can just use the @RooJpaEntity annotation.

   The JPA @Entity annotation is added to the Person_Roo_Jpa_Entity.aj ITD. This annotation marks the Person as persistable. By default, the JPA implementation of your choice will create a table definition in your database for this type. Once fields are added to the Person type, they will be added as columns to the Person table.

```
privileged aspect Person_Roo_Jpa_Entity {

    declare @type: Person: @Entity;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "id")
    private Long Person.id;

    @Version
    @Column(name = "version")
    private Integer Person.version;

    public Long Person.getId() {
        return this.id;
    }

    public void Person.setId(Long id) {
        this.id = id;
    }

    public Integer Person.getVersion() {
        return this.version;
    }

    public void Person.setVersion(Integer version) {
        this.version = version;
    }
}
```

As can be seen, the Person_Roo_Jpa_Entity.aj ITD introduces two fields by default. An *id* field (which is auto-incremented) and a *version* field (used for JPA-managed optimistic locking).

6. Person_Roo_Jpa_ActiveRecord.aj

   As mentioned previously, the @RooJpaActiveRecord annotation also triggers the creation of the Person_Roo_Jpa_ActiveRecord.aj ITD. This contains a number of persistence related CRUD methods into your Person type via the ITD:

```
privileged aspect Person_Roo_Jpa_ActiveRecord {

    @PersistenceContext
    transient EntityManager Person.entityManager;

    @Transactional
    public void Person.persist() {
        if (this.entityManager == null) this.entityManager = entityManager();
        this.entityManager.persist(this);
    }

    @Transactional
    public void Person.remove() {
        if (this.entityManager == null) this.entityManager = entityManager();
        if (this.entityManager.contains(this)) {
            this.entityManager.remove(this);
        } else {
            Person attached = this.entityManager.find(this.getClass(), this.id);
            this.entityManager.remove(attached);
        }
    }

    @Transactional
    public void Person.flush() {
        if (this.entityManager == null) this.entityManager = entityManager();
        this.entityManager.flush();
    }

    @Transactional
    public Person Person.merge() {
        if (this.entityManager == null) this.entityManager = entityManager();
        Person merged = this.entityManager.merge(this);
        this.entityManager.flush();
        return merged;
    }


    public static final EntityManager Person.entityManager() {
        EntityManager em = new Person().entityManager;
        if (em == null) throw new IllegalStateException("Entity manager has not been \
                        injected (is the Spring Aspects JAR configured as an AJC/AJDT \
                        aspects library?)");
        return em;
    }

    public static long Person.countPeople() {
        return entityManager().createQuery("select count(o) from Person o", Long.class)
```

```
                                        .getSingleResult();
    }

    @SuppressWarnings("unchecked")
    public static List<Person> Person.findAllPeople() {
        return entityManager().createQuery("select o from Person o", Person.class)
.getResultList();
    }

    public static Person Person.findPerson(Long id) {
        if (id == null) return null;
        return entityManager().find(Person.class, id);
    }

    @SuppressWarnings("unchecked")
    public static List<Person> Person.findPersonEntries(int firstResult, int
maxResults) {
        return entityManager().createQuery("select o from Person o", Person.class)
                    .setFirstResult(firstResult).setMaxResults(maxResults).getResultList
();
    }
}
```

The Person_Roo_Jpa_ActiveRecord.aj ITD introduces a number of methods such as *persist*(), *remove*(), *merge*(), *flush*() which allow the execution of ActiveRecord-style persistence operations on each Roo-managed JPA entity. Furthermore, a number of persistence-related convenience methods are provided. These methods are *countPeople*(), *findAllPeople*(), *findPerson*(..), and *findPersonEntries*(..).

All persistence methods are configured with Spring's Transaction support (Propagation.REQUIRED, Isolation.DEFAULT).

Similar to the @RooToString annotation you can change the default method name for all persistence-related methods generated through the @RooJpaActiveRecord annotation. For example:

```
@RooJpaActiveRecord(persistMethod = "save")
```

The entity jpa command offers a number of optional (but very useful) attributes worth mentioning. For example the **--testAutomatically** attribute can be used to have Roo to generate and maintain integration tests for the Person type (and the persistence methods generated as part of it). Furthermore, the **--abstract** and **--extends** attributes allow you to mark classes as abstract or inheritance patterns. Of course this can also be done directly in the Java sources of the Person type but sometimes it is useful to do this through a Roo command which can be scripted and replayed if desired. Other attributes allow you to define the identifier field name as well as the identifier field type which, in turn, allows the use of complex identifier types.

5.3. Field commands

As mentioned earlier in this chapter the field commands allow you to add pre-configured field definitions to your target entity type (Person.java in our example). In addition to simply adding the field names and types as defined via the command the appropriate JPA annotations are added to the field definitions. For example adding a birth day field to the Person.java type with the following command ...

```
~.Person roo> field date --fieldName birthDay --type java.util.Date
Managed SRC_MAIN_JAVA/com/foo/Person.java
Created SRC_MAIN_JAVA/com/foo/Person_Roo_JavaBean.aj
Managed SRC_MAIN_JAVA/com/foo/Person_Roo_ToString.aj
~.Person roo>
```

i. yields the following field definition in Person.java:

```
@Temporal(TemporalType.TIMESTAMP)
@DateTimeFormat(style = "M-")
private Date birthDay;
```

You'll notice that the @Temporal annotation is part of the JPA specification and defines how date values are persisted to and retrieved from the database in a transparent fashion. The @DateTimeFormat annotation is part of the Spring framework and takes care of printing and parsing Dates to and from String values when necessary (especially Web frontends frequently take advantage of this formatting capability).

Also note that Roo created a Person_Roo_JavaBean.aj ITD to generate accessors and mutators for the birthDay field and it also updated the toString() method to take the birthDay field into account.

Aside from the Date (and Calendar) type, the field command offers String, Boolean, Enum, Number, Reference and Set types. The Reference and Set types are of special interest here since they allow you to define relationships between your entities:

1. The field reference command will create a JPA many-to-one (default) or one-to-one relationship:

   ```
   ~.Person roo> field reference --fieldName car --type com.foo.Car
   ```

   The field definition added to the Person type contains the appropriate JPA annotations:

   ```
   @ManyToOne
   @JoinColumn
   private Car car;
   ```

The optional --cardinality command attribute allows you to define a one-to-one relationship (via JPAs @OneToOne annotation) between Person and Car if you wish:

```
@OneToOne
@JoinColumn
private Car car;
```

You can add the mappedBy attribute to the @OneToOne annotation to define the FK name handled by the inverse side (Car) of this relationship.

Consider the following constraint: when you delete a Person, any Car they have should also be deleted, but not vice versa (i.e. you should be able to delete a Car without deleting its owner). In the database, the foreign key should be in the "car" table.

```
@Entity
@RooJavaBean
@RooJpaActiveRecord
public class Person {

    // Inverse side ("car" table has the FK column)
    @OneToOne(cascade = CascadeType.ALL, mappedBy = "owner")
    private Car car;

}
```

```
@Entity
@RooJavaBean
@RooJpaActiveRecord
public class Car {

    // Owning side (this table has the FK column)
    @OneToOne
    @JoinColumn
    private Person owner;
}
```

If you delete a Person from the Person list, both the Person and the Car are deleted. So the cascading works. But if you delete a Car, the transaction will roll back and you will see an exception due it being referenced by a person. To overcome this situation you can add the following method to your Car.java:

```
@PreRemove
private void preRemove() {
    this.getOwner().setCar(null);
}
```

This hooks into the JPA lifecycle callback function and will set the reference between Person and Car to null before attempting to remove the record.

2. The field set command will allow you to create a many-to-many (default) or a one-to-many relationship:

```
field set --fieldName cars --type com.foo.Car
```

The field definition added to the Person type contains the appropriate JPA annotation:

```
@ManyToMany(cascade = CascadeType.ALL)
private Set<Car> cars = new HashSet<Car>();
```

To change the mapping type to one-to-many simply use the **--cardinality** attribute. To achieve a true m:n relationship you will need to issue the field set commands for both sides of the relationship.

Like the entity jpa command, the field command offeres a number of optional (but very useful) attributes worth mentioning. For example, you can change the field / column name translations with the **--column** attribute. Furthermore there are a number of attributes which translate directly to their equivalents defined in JSR 303 (Bean Validation). These attributes include **--notNull**, **--sizeMin**, **--sizeMax** and other related attributes. Please refer to the field command in the appendix to review the different attributes offered.

Chapter 6. Incremental Database Reverse Engineering (DBRE) Add-On

The incremental database reverse engineering (DBRE) add-on allows you to create an application tier of JPA 2.0 entities based on the tables in your database. DBRE will also incrementally maintain your application tier if you add or remove tables and columns.

6.1. Introduction

6.1.1. What are the benefits of Roo's incremental reverse engineering?

Traditional JPA reverse engineering tools are designed to introspect a database schema and produce a Java application tier once. Roo's incremental database reverse engineering feature differs because it has been designed to enable developers to repeatedly re-introspect a database schema and update their Java application. For example, consider if a column or table has been dropped from the database (or renamed). With Roo the re-introspection process would discover this and helpfully report errors in the Java tier wherever the now-missing field or entity was referenced. In simple terms, incremental database reverse engineering ensures Java type safety and easy application maintenance even if the database schema is constantly evolving. Just as importantly, Roo's incremental reverse engineering is implemented using the same unique design philosophy as the rest of Roo. This means very fast application delivery, clutter-free .java source files, extensive usability features in the shell (such as tab completion and hinting) and so on.

6.1.2. How does DBRE work?

6.1.2.1. Obtaining database metadata

The DBRE commands (see Section 9.3, "DBRE Add-On commands" below) make live connections to the database configured in your Roo project and obtain database metadata from the JDBC driver's implementation of the standard java.sql.DatabaseMetadata interface. When the database is reverse engineered, the metadata information is converted to XML and is stored and maintained in the dbre.xml file in the src/main/resources directory of your project. DBRE creates JPA entities based on the table names in your database and fields based on the column names in the tables. Simple and composite primary keys are supported (see Section 9.5.2, "Composite primary keys" for more details) and relationships between entities are also created using the imported and exported key information obtained from the metadata.

6.1.2.2. Class and field name creation

DBRE creates entity classes with names that are derived from the associated table name using a simple algorithm. If a table's name contains an underscore, hyphen, forward or back slash character, an upper case letter is substituted for each of these characters. This is also similar for column and field names. The following tables contain some examples.

Table name	DBRE-produced entity class name
order	Order.java
line_item	LineItem.java
EAM_MEASUREMENT_DATA_1H	EamMeasurementData1h.java
COM-FOO\BAR	ComFooBar.java

Column name	DBRE-produced field name
order	order
EMPLOYEE_NUMBER	employeeNumber
USR_CNT	usrCnt

6.2. Installation

DBRE supports most of the relational databases that can be configured for Roo-managed projects such as MySQL, MS SQL, and PostgreSQL. These drivers are auto-detected by Roo and you will be prompted by the Roo shell to download your configured database's JDBC driver when you first issue the database introspect or database reverse engineer commands (see Section 9.3, "DBRE Add-On commands" below). For example, if you have configured your Roo project to use a PostgreSQL database, when the database introspect command is first issued, you will see the following console output:

```
roo> database introspect --schema unable-to-obtain-connection
Searching 'org.postgresql.Driver' on installed repositories
1 matches found with 'org.postgresql.Driver' on installed repositories
ID   BUNDLE SYMBOLIC NAME                                DESCRIPTION
-------------------------------------------------------------------------------
01   org.springframework.roo.wrapping.postgresql-jdbc3   Spring Roo - Wrapping -
postgresql-jdbc3
-------------------------------------------------------------------------------
[HINT] use 'addon info bundle --bundleSymbolicName' to see details about a search result
[HINT] use 'addon install bundle --bundleSymbolicName' to install a specific add-on
version
Located add-on that may offer this JDBC driver
JDBC driver not available for 'org.postgresql.Driver'
```

You can get further information about the search result with the following command:

```
roo>  addon info bundle --bundleSymbolicName org.springframework.roo.wrapping.postgresql-
jdbc3
```

This may list several versions of a driver if available.

You can then install the latest Postgres JDBC driver by entering the following Roo command:

```
roo>  addon install bundle --bundleSymbolicName
org.springframework.roo.wrapping.postgresql-jdbc3
```

The JDBC driver for Postgres is immediately available for you to use. You can now enter the database introspect and database reverse engineer commands (see Section 9.3, “DBRE Add-On commands” below).

Note: currently there are no open-source JDBC drivers for Oracle or DB2 and Roo does not provide OSGi drivers for these databases. If you are an Oracle or DB2 user, you will need to obtain an OSGi-enabled driver from Oracle or IBM respectively or wrap your own Oracle or DB2 driver jars using Roo's wrapping facility. Use the addon create wrapper to turn an existing Oracle JDBC driver into an OSGi bundle you can install into Roo. Roo does provide a wrapping pom.xml for the DB2 Express-C edition that can be used to convert your db2jcc4.jar into an OSGi-compliant driver. You can then use the osgi start command to install the jar, for example:

```
roo> addon install url --url file:///tmp/org.springframework.roo.wrapping.db2jcc4-
9.7.2.0001.jar
```

6.3. DBRE Add-On commands

After you have configured your persistence layer with the jpa setup command and installed all the JDBC drivers, you can introspect and reverse engineer the database configured for your project. DBRE contains two commands:

1

```
roo> <strong>database introspect --schema</strong> --file --enableViews
```

This command displays the database structure, or schema, in XML format. The --schema is mandatory and for databases which support schemas, you can press tab to display a list of schemas from your database. You can use the --file option to save the information to the specified file.

The --enableViews option when specified will also retrieve database views and display them with the table information.

Notes:

- The term "schema" is not used by all databases, such as MySQL and Firebird, and for these databases the target database name is contained in the JDBC URL connection string. However the

--schema option is still required but Roo's tab assist feature will display "no-schema-required".

- PostgreSQL upper case schema names are not supported.

2

```
roo> <strong>database reverse engineer --schema</strong> --package --activeRecord
--repository
--service --testAutomatically --enableViews
--includeTables --excludeTables
--includeNonPortableAttributes
--disableVersionFields --disableGeneratedIdentifiers
```

This command creates JPA entities in your project representing the tables and columns in your database. As for the database introspect command, the --schema option is required and tab assistance is available. You can use the --package option to specify a Java package where your entities will be created. If you do not specify the --package option on second and subsequent executions of the database reverse engineer command, new entities will be created in the same package as they were previously created in.

Use the --activeRecord option to create 'Active Record' entities (default if not specified).

Use the --repository option to create Spring Data JPA Repositories for each entity. If specified as true, the --activeRecord option is ignored.

Use the --service option to create a service layer for each entity.

Use the --testAutomatically option to create integration tests automatically for each new entity created by reverse engineering.

The --enableViews option when specified will also retrieve database views and reverse engineer them into entities. Note that this option should only be used in specialised use cases only, such as those with database triggers.

You can generate non-portable JPA @Column attributes, such as 'columnDefinition' by specifying the --includeNonPortableAttributes option.

Use the --disableVersionFields option to disable the generation of 'version' fields.

Use the --disableGeneratedIdentifiers option to disable auto generated identifiers.

Since the DBRE Add-on provides incremental database reverse engineering, you can execute the command as many times as you want and your JPA entities will be maintained by Roo, that is, new fields will be added if new columns are added to a table, or fields will be removed if columns are deleted. Entities are also deleted in certain circumstances if their corresponding tables are dropped.

Examples of the database reverse engineer command:

- `roo> database reverse engineer --schema order --package ~.domain --excludeTables "foo* bar?"`

  This will reverse engineer all tables *except* any table whose name starts with 'foo' and any table called bar with one extra character, such as 'bar1' or 'bars'.

  You can use the --includeTables and --excludeTables option to specify tables that you want or do not want reverse engineered respectively. The options can take one or more table names. If more than one table is required, the tables must be enclosed in double quotes and each separated by a space. Wild-card searching is also permitted using the asterisk (*****) character to match one or more characters or the '?' character to match exactly one character. For example:

  Note: excluding tables not only prevent entities from being created but associations are also not created in other entities. This is done to prevent compile errors in the source code.

- `roo> database reverse engineer --schema order --package ~.domain --includeTables "foo* bar?"`

  This will reverse engineer all tables who table whose name starts with 'foo' and any table called bar with one extra character, such as 'bar1' or 'bars'.

- You can also reverse engineer more than one schema by specifying a doubled-quoted space-separated list of schemas. Reverse engineering of foreign-key releationships between tables in different schemas is supported. For example:

  ```
  roo> database reverse engineer --schema "schema1 schema2 schema3" --package ~.domain
  ```

  This will reverse engineer all tables from schemas "schema1", "schema2", and "schema3".

6.4. The @RooDbManaged annotation

The @RooDbManaged annotation is added to all new entities created by executing the database reverse engineer command. Other Roo annotations, @RooJpaActiveRecord, @RooJavaBean, and @RooToString are also added to the entity class. The attribute "automaticallyDelete" is added to the @RooDbManaged annotation and is set to "true" so that Roo can delete the entity if the associated table has been dropped. However, if "automaticallyDelete" is set to "false", or if any annotations, fields, constructors, or methods have been added to the entity (i.e in the .java file), or if any of the Roo annotations are removed, the entity will not be deleted.

The presence of the @RooDbmanaged annotation on an entity class triggers the creation of an AspectJ inter-type declaration (ITD) ".aj" file where fields and their getters and setters are stored matching the columns in the table. For example, if an entity called Employee.java is created by the database reverse engineer command, a file called Employee_Roo_DbManaged.aj is also created and maintained by Roo. All the columns of the matching employee table will cause fields to be created in the entity's DbManaged ITD. An example of a DBRE-created entity is as follows:

```
@RooJavaBean
@RooToString
@RooDbManaged(automaticallyDelete = true)
@RooJpaActiveRecord(table = "employee", schema = "expenses")
public class Employee {
}
```

Along with the standard entity, toString, configurable ITDs, a DbManaged ITD is created if there are more columns in the employee table apart from a primary key column. For example, if the employee table has mandatory employee name and employee number columns, and a nullable age column the ITD could look like this:

```
privileged aspect Employee_Roo_DbManaged {

    @Column(name = "employee_number")
    @NotNull
    private String Employee.employeeNumber;

    public String Employee.getEmployeeNumber() {
        return this.employeeNumber;
    }

    public void Employee.setEmployeeNumber(String employeeNumber) {
        this.employeeNumber = employeeNumber;
    }

    @Column(name = "employee_name", length = "100")
    @NotNull
    private String Employee.employeeName;

    public String Employee.getEmployeeName() {
        return this.employeeName;
    }

    public void Employee.setEmployeeName(String employeeName) {
        this.employeeName = employeeName;
    }

    @Column(name = "age")
    private Integer Employee.age;

    public Integer Employee.getAge() {
        return this.age;
    }

    public void Employee.setAge(Integer age) {
        this.age = age;
    }

    ...
}
```

If you do not want DBRE to manage your entity any more, you can "push-in" refactor the fields and methods in the DbManaged ITD and remove the @RooDbManaged annotation from the .java file.

6.5. Supported JPA 2.0 features

DBRE will produce and maintain primary key fields, including composite keys, entity relationships

such as many-valued and single-valued associations, and other fields annotated with the JPA @Column annotation.

The following sections describe the features currently supported.

6.5.1. Simple primary keys

For a table with a single primary key column, DBRE causes an identifier field to be created in the entity ITD annotated with @Id and @Column. This is similar to executing the entity jpa command by itself.

6.5.2. Composite primary keys

For tables with two or more primary key columns, DBRE will create a primary key class annotated with @RooIdentifier(dbManaged = true) and add the "identifierType" attribute with the identifier class name to the @RooJpaActiveRecord annotation in the entity class. For example, a line_item table has two primary keys, line_item_id and order_id. DBRE will create the LineItem entity class and LineItemPK identifier class as follows:

```
@RooJavaBean
@RooToString
@RooDbManaged(automaticallyDelete = true)
@RooJpaActiveRecord(identifierType = LineItemPK.class, table = "line_item", schema =
"order")
public class LineItem {
}
```

```
@RooIdentifier(dbManaged = true)
public class LineItemPK {
}
```

Roo will automatically create the JPA entity ITD containing a field annotated with @EmbeddedId with type LineItemPK as follows:

```
privileged aspect LineItem_Roo_JpaEntity {

    declare @type: LineItem: @Entity;

    declare @type: LineItem: @Table(name = "line_item", schema = "order");

    @EmbeddedId
    private LineItemPK LineItem.id;

    public LineItemPK LineItem.getId() {
        return this.id;
    }

    public void LineItem.setId(LineItemPK id) {
        this.id = id;
    }

    ...
}
```

and an identifier ITD for the LineItemPK class containing the primary key fields and the type annotation for @Embeddable, as follows:

```
privileged aspect LineItemPK_Roo_Identifier {

    declare @type: LineItemPK: @Embeddable;

    @Column(name = "line_item_id", nullable = false)
    private BigDecimal LineItemPK.lineItemId;

    @Column(name = "order_id", nullable = false)
    private BigDecimal LineItemPK.orderId;

    public LineItemPK.new(BigDecimal lineItemId, BigDecimal orderId) {
        super();
        this.lineItemId = lineItemId;
        this.orderId = orderId;
    }

    private LineItemPK.new() {
        super();
    }

    ...
}
```

If you decide that your table does not require a composite primary key anymore, the next time you execute the database reverse engineer command, Roo will automatically change the entity to use a single primary key and remove the identifier class if it is permitted.

6.5.3. Entity relationships

One of the powerful features of DBRE is its ability to create relationships between entities automatically based on the foreign key information in the dbre.xml file. The following sections describe the associations that can be created.

6.5.3.1. Many-valued associations with many-to-many multiplicity

Many-to-many associations are created if a join table is detected by DBRE. To be identified as a many-to-many join table, the table must have exactly two primary keys and have exactly two foreign-keys pointing to other entity tables and have no other columns.

For example, the database contains a product table and a supplier table. The database has been modelled such that a product can have many suppliers and a supplier can have many products. A join table called product_supplier also exists and links the two tables together by having a composite primary key made up of the product id and supplier id and foreign keys pointing to each of the primary keys of the product and supplier tables. DBRE will create a bi-directional many-to-many association. DBRE will designate which entities are the owning and inverse sides of the association respectively and annotate the fields accordingly as shown in the following code snippets:

```
privileged aspect Product_Roo_DbManaged {

    @ManyToMany
    @JoinTable(name = "product_supplier",
        joinColumns = {
            @JoinColumn(name = "prod_id") },
        inverseJoinColumns = {
            @JoinColumn(name = "supp_id") })
    private Set<Supplier> Product.suppliers;

    ...
}
```

```
privileged aspect Supplier_Roo_DbManaged {

    @ManyToMany(mappedBy = "suppliers")
    private Set<Product> Supplier.products;

    ...
}
```

DBRE will also create many-to-many associations where the two tables each have composite primary keys. For example:

```
privileged aspect Foo_Roo_DbManaged {

    @ManyToMany
    @JoinTable(name = "foo_bar",
        joinColumns = {
            @JoinColumn(name = "foo_bar_id1", referencedColumnName = "foo_id1"),
            @JoinColumn(name = "foo_bar_id2", referencedColumnName = "foo_id2") },
        inverseJoinColumns = {
            @JoinColumn(name = "foo_bar_id1", referencedColumnName = "bar_id1"),
            @JoinColumn(name = "foo_bar_id2", referencedColumnName = "bar_id2") })
    private Set<Bar> Foo.bars;

    ...
}
```

6.5.3.2. Single-valued associations to other entities that have one-to-one multiplicity

If the foreign key column represents the entire primary key (or the entire index) then the relationship between the tables will be one to one and a bi-directional one-to-one association is created.

For example, the database contains a customer table and an address table and a customer can only have one address. The following code snippets show the one-to-one mappings:

```
privileged aspect Address_Roo_DbManaged {

    @OneToOne
    @JoinColumn(name = "address_id")
    private Party Address.customer;

    ...
}
```

```
privileged aspect Customer_Roo_DbManaged {

    @OneToOne(mappedBy = "customer")
    private Address Party.address;

    ...
}
```

6.5.3.3. Many-valued associations with one-to-many multiplicity

If the foreign key column is part of the primary key (or part of an index) then the relationship between the tables will be one to many. An example is shown below:

```
privileged aspect Order_Roo_DbManaged {

    @OneToMany(mappedBy = "order")
    private Set<LineItem> Order.lineItems;

    ...
}
```

6.5.3.4. Single-valued associations to other entities that have many-to-one multiplicity

When a one-to-many association is created, for example a set of LineItem entities in the Order entity in the example above, DBRE will also create a corresponding many-to-one association in the LineItem entity, as follows:

```
privileged aspect LineItem_Roo_DbManaged {

    @ManyToOne
    @JoinColumn(name = "order_id", referencedColumnName = "order_id")
    private Order LineItem.order;

    ...
}
```

6.5.3.5. Multiple associations in the same entity

DBRE will ensure field names are not duplicated. For example, if an entity has more than one association to another entity, the field names will be created with unique names. The following code snippet illustrates this:

```
privileged aspect Foo_Roo_DbManaged {

    @ManyToMany
    @JoinTable(name = "foo_bar",
        joinColumns = {
            @JoinColumn(name = "foo_bar_id1", referencedColumnName = "foo_id1"),
            @JoinColumn(name = "foo_bar_id2", referencedColumnName = "foo_id2") },
        inverseJoinColumns = {
            @JoinColumn(name = "foo_bar_id1", referencedColumnName = "bar_id1"),
            @JoinColumn(name = "foo_bar_id2", referencedColumnName = "bar_id2") })
    private Set<Bar> Foo.bars;

    @ManyToMany
    @JoinTable(name = "foo_com",
        joinColumns = {
            @JoinColumn(name = "foo_com_id1", referencedColumnName = "foo_id1"),
            @JoinColumn(name = "foo_com_id2", referencedColumnName = "foo_id2") },
        inverseJoinColumns = {
            @JoinColumn(name = "foo_com_id1", referencedColumnName = "bar_id1"),
            @JoinColumn(name = "foo_com_id2", referencedColumnName = "bar_id2") })
    private Set<Bar> Foo.bars1;

    ...
}
```

6.5.4. Other fields

DBRE will detect column types from the database metadata and create and maintain fields and field annotations appropriately. Strings, dates, booleans, numeric fields, CLOBs and BLOBs are all supported by DBRE, as well as the JSR 303 @NotNull validation constraint.

6.5.5. Existing fields

Roo checks the .java file for a field before it creates it in the ITD. If you code a field in the entity's .java file, Roo will not create the field in the DbManaged ITD if detected in the database metadata. For example, if your table has a column called 'name' and you have added a field called 'name' to the .java file, Roo will not create this field in the ITD when reverse engineered.

Roo also ensures the entity's identity field is unique. For example if the @Id field is called 'id' but you also add a field with the same name to the .java file, DBRE will automatically rename the @Id field by prefixing it with an underscore character.

6.6. Troubleshooting

This section explains scenarios that may be encountered when using the DBRE feature.

- **Executing the database introspect or database reverse engineer commands causes the message 'JDBC driver not available for oracle.jdbc.OracleDriver' to be displayed**

  This is due to the Oracle JDBC driver not having been installed. The driver must be installed if you have installed Roo for the first time. See Section 9.2, “Installation”. This also applies to other databases, for example, HSQL and H2.

- **Executing the database introspect or database reverse engineer commands with the Firebird database configured causes the message 'Exception in thread "JLine Shell" java.lang.NoClassDefFoundError: javax/resource/ResourceException' to be displayed**

  This is due to the javax.resource connector jar not installed. Remove the cache directory under your Roo installation directory, start the Roo shell, and run the command:

  ```
  addon install url --url
          http://spring-roo-
  repository.springsource.org/release/org/springframework/roo/wrapping/org.springframewor
  k.roo.wrapping.connector/1.0.0010/org.springframework.roo.wrapping.connector-
  1.0.0010.jar
  ```

  Re-install the Firebird driver. See Section 9.2, “Installation”.

- **The error message 'Caused by: org.hibernate.HibernateException: Missing sequence or table: hibernate_sequence' appears when starting Tomcat**

  When the database reverse engineer command is first run, the property determining whether tables are created and dropped which is defined in the persistence.xml file is modified to a value that prevents new database artifacts from being created. This is done to avoid deleting the data in your tables when unit tests are run or a web application is started. For example, if you use Hibernate as your JPA 2.0 provider the property is called 'hibernate.hbm2ddl.auto' and is initially set to 'create' when the project is first created. This value causes Hibernate to create tables and sequences and allows you to run unit tests and start a web application. However, the property's value is changed to 'validate' when the database reverse engineer command is executed. Other JPA providers such as EclipseLink and OpenJPA have a similar property which are also changed when the command is run. If you see this issue when running unit tests or when starting your web application after reverse engineering, you may need to change the property back to 'create' or 'update'. Check your persistence.xml for the property values for other JPA providers.

- **The message 'Unable to maintain database-managed entity**<*entity name*>**because its associated table name could not be found' appears in the Roo console during reverse engineering**

  When DBRE first creates an entity it puts in the table name in the 'table' attribute of the @RooJpaActiveRecord annotation. This is the only mechanism DBRE has for associating an entity with a table. If you remove the 'table' attribute, DBRE has no way of determining what the entity's corresponding table is and as a result cannot maintain the entity's fields and associations.

Chapter 7. Application Layering

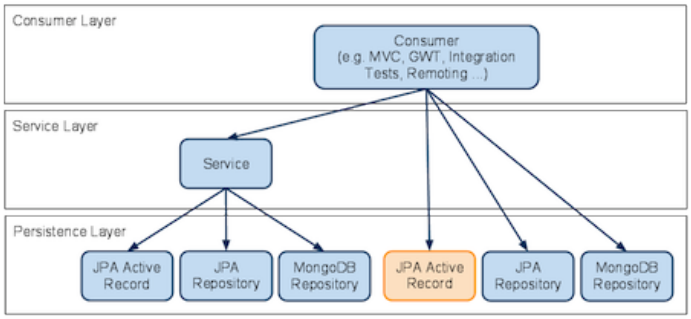
Java enterprise applications can take many shapes and forms depending on their requirements. Depending on these requirements, you need to decide which layers your application needs. Spring Roo team recommends to generate applications using entity layer, web layer, service layer and repository layer.

This section provides an overview of Roo's support for service and repository layers.

> Note: This section provides an overview of the application layering options Spring Roo offers since the **1.2.0.M1** release. It does not discuss the merits of different approaches to architecting enterprise applications.

7.1. The Big Picture

With the Roo 1.2.0 release internals have been changed to allow the integration of multiple application layers. This is particularly useful for the support of different persistence mechanisms. In previous releases the only persistence supported in Roo core was the JPA Entity Active Record pattern. With the internal changes in place Roo can now support alternative persistence providers which support application layering.



While there are a number of new layering and persistence choices available, by default Roo will continue to support the JPA Active Record Entity by default (marked orange). However, you can easily change existing applications by adding further service or repository layers (details below). If you add new layers Roo will automatically change its ITDs in the consumer layer or service layer respectively. For example it will automatically inject and call a new service layer within an existing MVC controller, Integration test or data on demand for a given domain type.

7.2. Persistence Layers

There are now three options available in Roo core to support data persistence, JPA Entities (Active Record style) and JPA Repositories

7.2.1. JPA Entities (Active Record style)

Active record-style JPA Entities have been the default since the first release of Spring Roo and will remain so. In order to configure your project for JPA persistence, you can run the jpa setup command:

```
roo> <strong>jpa setup</strong> --provider HIBERNATE --database HYPERSONIC_PERSISTENT
```

This configures your project to use the Hibername object relational mapper along with a in-memory database (HSQLDB). Further details about this persistence option can be found here.

Active record-style JPA entities supported by Roo need to have a **@RooJpaActiveRecord** annotation which takes care of providing an ID field along with its accessor and mutator, In addition this annotation creates the typical CRUD methods to support data access.

```
roo> <strong>entity jpa</strong> --class ~.domain.Pizza
```

This command will create a Pizza domain type along with active record-style methods to persist, update, read and delete your entity. The following example also contains a number of fields which can be added through the field command via the Roo shell.

```
@RooJavaBean
@RooToString
@RooJpaActiveRecord
public class Pizza {

    @NotNull
    @Size(min = 2)
    private String name;

    private BigDecimal price;

    @ManyToMany(cascade = CascadeType.ALL)
    private Set<Topping> toppings = new HashSet<Topping>();

    @ManyToOne
    private Base base;
}
```

Further details about command options and functionalities provided by active record-style JPA Entities please refer to the Persistence Add-on chapter.

7.2.2. JPA Entities (Repository style)

Developers who require a repository / DAO layer instead of the default Roo entity-based persistence approach can do so by creating a Spring Data JPA backed repository for a given JPA domain type. The domain type backing the repository needs have a JPA @Entity annotation and also a ID field defined along with accessors and mutators. After configuring your project for JPA persistence via the jpa setup command, this functionality is automatically provided by annotating the domain type with Roo's **@RooJpaEntity** annotation.

```
roo> <strong>entity jpa</strong> --class ~.domain.Pizza <strong>--activeRecord
false</strong>
```

By defining --activeRecord false you can opt out of the otherwise default Active Record style. The following example also contains a number of fields which can be added through the field command via the Roo shell.

```
@RooJavaBean
@RooToString
@RooJpaEntity
public class Pizza {

    @NotNull
    @Size(min = 2)
    private String name;

    private BigDecimal price;

    @ManyToMany(cascade = CascadeType.ALL)
    private Set<Topping> toppings = new HashSet<Topping>();

    @ManyToOne
    private Base base;
}
```

With a domain type in place you can now create a new repository for this type by using the repository jpa command:

```
roo> <strong>repository jpa</strong> --interface ~.repository.PizzaRepository --entity
~.domain.Pizza
```

This will create a simple interface definition which leverages Spring Data JPA:

```
@RooJpaRepository(domainType = Pizza.class)
public interface PizzaRepository {
}
```

Of course, you can simply add the **@RooJpaRepository** annotation on any interface by hand in lieu of issuing the repository jpa command in the Roo shell.

The adition of the **@RooJpaRepository** annotation will trigger the creation of a fairly trivial AspectJ ITD which adds an extends statement to the PizzaRepository interface resulting in the equivalent of this interface definition:

```
public interface PizzaRepository extends JpaRepository<Pizza, Long> {}
```

Note, the JpaRepository interface is part of the Spring Data JPA API and provides all CRUD functionality out of the box.

7.3. Service Layer

Developers can also choose to create a service layer in their application. By default, Roo will create a service interface (and implementation) for one or more domain entities. If a persistence-providing layer such as Roo's default entity layer or a repository layer is present for a given domain entity, the service layer will expose the CRUD functionality provided by this persistence layer through its interface and implementation.

As per Roo's conventions all functionality will be introduced via AspectJ ITDs therefore providing the developer a clean canvas for implementing custom business logic which does not naturally fit into domain entities. Other common use cases for service layers are security or integration of remoting such as Web Services. For a more detailed discussion please refer to the architecture chapter.

The integration of a services layer into a Roo project is similar to the repository layer by using the @RooService annotation directly or the service command in the Roo shell:

```
roo> <strong>service type</strong> --interface ~.service.PizzaService --entity
~.domain.Pizza
```

This command will create the PizzaService interface in the defined package and additionally a PizzaServiceImpl in the same package (the name and package of the PizzaServiceImpl can be customized via the optional --class attribute).

```
@RooService(domainTypes = { Pizza.class })
public interface PizzaService {
}
```

Following Roo conventions the default CRUD method definitions can be found in the ITD:

```
void savePizza(Pizza pizza);
Pizza findPizza(Long id);
List<Pizza> findAllPizzas();
List<Pizza> findPizzaEntries(int firstResult, int maxResults);
long countAllPizzas();
Pizza updatePizza(pizza pizza);
void deletePizza(Pizza pizza);
```

Similarly, the PizzaServiceImpl is rather empty:

```
public class PizzaServiceImpl implements PizzaService {
}
```

Through the AspectJ ITD the PizzaServiceImpl type is annotated with the **@Service** and **@Transactional** annotations by default. Furthermore the ITD will introduce the following methods and fields into the target type:

```
@Autowired PizzaRepository pizzaRepository;

public void savePizza(Pizza pizza) {
    pizzaRepository.save(pizza);
}

public Pizza findPizza(Long id) {
    return pizzaRepository.findOne(id);
}

public List<Pizza> findAllPizzas() {
    return pizzaRepository.findAll();
}

public List<Pizza> findPizzaEntries(int firstResult, int maxResults) {
    return pizzaRepository.findAll(new PageRequest(firstResult / maxResults, maxResults)
).getContent();
}

public long countAllPizzas() {
    return pizzaRepository.count();
}

public Pizza updatePizza(Pizza pizza) {
    return pizzaRepository.save(pizza);
}

public void deletePizza(Pizza pizza) {
    pizzaRepository.delete(pizza);
}
```

As you can see, Roo will detect if a persistence provider layer exists for a given domain type and automatically inject this component in order to delegate all service layer calls to this layer. In case no persistence (or other 'lower level' layer exists, the service layer ITD will simply provide method stubs.

Chapter 8. Web MVC Add-On

> *CSS considerations*: The Web UI has been tested successfully with FireFox, Opera, Safari, Chrome, and IE. Given that IE6 is not supported any more by most players in the market, it has a number of severe technical limitations and it has a fast declining user base Spring Roo does not support IE6. Your mileage may vary - there will likely be issues with CSS support.

The Web MVC add-ons allow you to conveniently scaffold Spring MVC controllers and JSP(X) views for an existing domain model. Currently this domain model is derived from the Roo supported JPA integration through the entity jpa and related field commands. As shown in the Introduction and the Beginning With Roo: The Tutorial the Web MVC scaffolding can deliver a fully functional Web frontend to your domain model. The following features are included:

- Automatic update of JSPX view artifacts reflecting changes in the domain model
- A fully customizable set JSP of tags is provided, all tags are XML only (no tag-backing Java source code is required)
- Tags offer integration with the Dojo Ajax toolkit for client-side validation, date pickers, tool tips, filtering selects etc
- Automatic URL rewriting to provide best-practice RESTful URIs
- Integration of Apache Tiles templating framework to allow for structural customization of the Web user interface
- Use of cascading stylesheets to allow for visual customization of the Web user interface
- Use of Spring MVC themeing support to dynamically adjust Web user interface by changing CSS
- Internationalization of complete Web UI is supported by simply adding new message bundles (6+ languages are already suppprted)
- Pagination integration for large datasets
- Client- and server-side validation based on JSR 303 constraints defined in the domain layer
- Generated controllers offer best-practice use of Spring framework MVC support

The following sections will offer further details about available commands to generate Web MVC artifacts and also the new JSP(X) round-tripping model introduced in Roo 1.1.

8.1. Controller commands

The Web MVC addon offers a number of commands to generate and maintain various Web artifacts:

- `~.Person roo> web mvc setup`

  The first time the web mvc setup command is executed Roo will install all artifacts required for the Web UI.

- `~.Person roo> web mvc scaffold --class com.foo.web.PersonController`

  The controller scaffold command will create a Spring MVC controller for the Person entity with the following method signatures:

Method Signature	Comment
public String **create**(@Valid Person person, BindingResult result, ModelMap modelMap) \{..}	The create method is triggered by HTTP POST requests to /<app-name>/people. The submitted form data will be converted to a Person object and validated against JSR 303 constraints (if present). Response is redirected to the show method.
public String **createForm**(ModelMap modelMap) \{..}	The create form method is triggered by a HTTP GET request to /<app-name>/people?form. The resulting form will be prepopulated with a new instance of Person, referenced Cars and datepatterns (if needed). Returns the Tiles view name.
public String **show**(@PathVariable("id") Long id, ModelMap modelMap) \{..}	The show method is triggered by a HTTP GET request to /<app-name>/people/<id>. The resulting form is populated with a Person instance identifier by the id parameter. Returns the Tiles view name.
public String **list**(@RequestParam(value = "page", required = false) Integer page, @RequestParam(value = "size", required = false) Integer size, ModelMap modelMap) \{..}	The list method is triggered by a HTTP GET request to /<app-name>/people. This method has optional parameters for pagination (page, size). Returns the Tiles view name.
public String **update**(@Valid Person person, BindingResult result, ModelMap modelMap) \{..}	The update method is triggered by a HTTP PUT request to /<app-name/people. The submitted form data will be converted to a Person object and validated against JSR 303 constraints (if present). Response is redirected to the show method.

Method Signature	Comment
public String **updateForm**(@PathVariable("id") Long id, ModelMap modelMap) \{	The update form method is triggered by a HTTP GET request to /<app-name>/people/<id>?form. The resulting form will be prepopulated with a Person instance identified by the id parameter, referenced Cars and datepatterns (if needed). Returns the Tiles view name.
public String **delete**(@PathVariable("id") Long id, @RequestParam(value = "page", required = false) Integer page, @RequestParam(value = "size", required = false) Integer size) \{..}	The delete method is triggered by a HTTP DELETE request to /<app-name>/people/<id>. This method has optional parameters for pagination (page, size). Response is redirected to the list method.
public Collection<Car> **populateCars**() \{..}	This method prepopulates the 'car' attribute. This method can be adjusted to handle larger collections in different ways (pagination, caching, etc).
void **addDateTimeFormatPatterns**(ModelMap modelMap) \{..}	Method to register date and time patterns used for date and time binding for form submissions.

As you can see Roo implements a number of methods to offer a RESTful MVC frontend to your domain layer. All of these methods can be found in the PersonController_Roo_Controller.aj ITD. Feel free to push-in any (or all) of these methods to change default behaviour implemented by Roo.

The web mvc scaffold command offers a number of optional attributes which let you refine the way paths are managed and which methods should be generated in the controller. The **--disallowedOperations** attribute helps you refine which methods should not be generated in the scaffolded Roo controller. If you want to prevent several methods from being generated provide a comma-separated list (i.e.: --disallowedOperations delete,update,create). You can also specify which methods should be generated and which not in the PersonController.java source:

```
@RooWebScaffold(path = "people", formBackingObject = Person.class, create = false,
update = false, delete = false)
@RequestMapping("/people")
@Controller
public class PersonController {}
```

If you don't define a custom path Roo will use the plural representation of the simple name of the

form backing entity (in our case 'people'). If you wish you can define more complex custom paths like /public/people or /my/special/person/uri (try to to stick to REST patterns if you can though). A good use case for creating controllers which map to custom paths is security. You can, for example create two controllers for the Person entity. One with the default path (/people) for public access (possibly with delete, and update functionality disabled) and one for admin access (/admin/people). This way you can easily secure the /admin/* path with the Spring Security addon.

- `roo> web mvc all --package ~.web`

  The web mvc all command provides a convenient way to quickly generate Web MVC controllers for all JPA entities Roo can find in your project. You need to specify the **--package** attribute to define a package where these controllers should be generated. While the web mvc all command is convenient, it does not give you the same level of control compared to the web mvc scaffold command.

- `roo> web mvc controller --class com.foo.web.CarController --preferredMapping /public/car`

```
Created SRC_MAIN_JAVA/com/foo/web/CarController.java
Created SRC_MAIN_WEBAPP/WEB-INF/views/public/car
Created SRC_MAIN_WEBAPP/WEB-INF/views/public/car/index.jspx
Managed SRC_MAIN_WEBAPP/WEB-INF/i18n/application.properties
Managed SRC_MAIN_WEBAPP/WEB-INF/views/menu.jspx
Created SRC_MAIN_WEBAPP/WEB-INF/views/public/car/views.xml
```

  The web mvc controller command is different from the other two controller commands shown above. It does *not* generate an ITD with update, create, delete and other methods to integrate with a specific form backing entity. Instead, this command will create a simple controller to help you get started for developing a custom functionality by stubbing a simple *get*(), *post*() and *index*() method inside the controller:

```
@RequestMapping("/public/car/**")
@Controller
public class CarController {

    @RequestMapping
    public void get(ModelMap modelMap, HttpServletRequest request,
                    HttpServletResponse response) {
    }

    @RequestMapping(method = RequestMethod.POST, value = "{id}")
    public void post(@PathVariable Long id, ModelMap modelMap, HttpServletRequest
request,
                     HttpServletResponse response) {
    }

    @RequestMapping
    public String index() {
        return "public/car/index";
    }
}
```

In addition, this controller is registered in the Web MVC menu and the application Tiles definition. Furthermore, a simple view (under WEB-INF/views/public/car/index.jspx).

- `roo> web mvc finder add --class ~.web.PersonController --formBackingType ~.domain.Person`

  The web mvc finder add command used from the Roo shell will introdroduce the **@RooWebFinder** annotation into the specified target type.

- `roo> web mvc finder all`

  The web mvc finder all command used from the Roo shell will introdroduce the **@RooWebFinder** annotations to all existing controllers which have a form backing type that offers dynamic finders.

8.2. Application Conversion Service

Whenever a controller is created for the first time in an application, Roo will also install an application-wide ConversionService and configure it for use in webmvc-config.xml as follows:

```
<mvc:annotation-driven conversion-service="applicationConversionService"/>
...
<bean id="applicationConversionService" class=
"com.springsource.vote.web.ApplicationConversionServiceFactoryBean"/>
```

Spring MVC uses the ConversionService when it needs to convert between two objects types — e.g. Date

and String. To become more familiar with its features we recommend that you review the (brief) sections on "Type Conversion" and "Field Formatting" in the Spring Framework documentation.

The ApplicationConversionServiceFactoryBean is a Roo-managed Java class and it looks like this:

```
@RooConversionService
public class ApplicationConversionServiceFactoryBean extends
FormattingConversionServiceFactoryBean {

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

}
```

As the comment indicates you can use the installFormatters() method to register any Converters and Formatters you wish to add. In addition to that Roo will automatically maintain an ITD with Converter registrations for every associated entity that needs to be displayed somewhere in a view. A typical use case is where entities from a many-to-one association need to be displayed in one of the JSP views. Rather than using the toString() method for that, a Converter defines the formatting logic for how to present the associated entity as a String.

Note, a custom written or pushed in converter method needs to be registered manually via the *installFormatters* method which is already present in your *ApplicationConversionServiceFactoryBean.java* source code.

In some cases you may wish to customize how a specific entity is formatted as a String in JSP views. For example suppose we have an entity called Vote. To customize how it is displayed in the JSP views add a method like this:

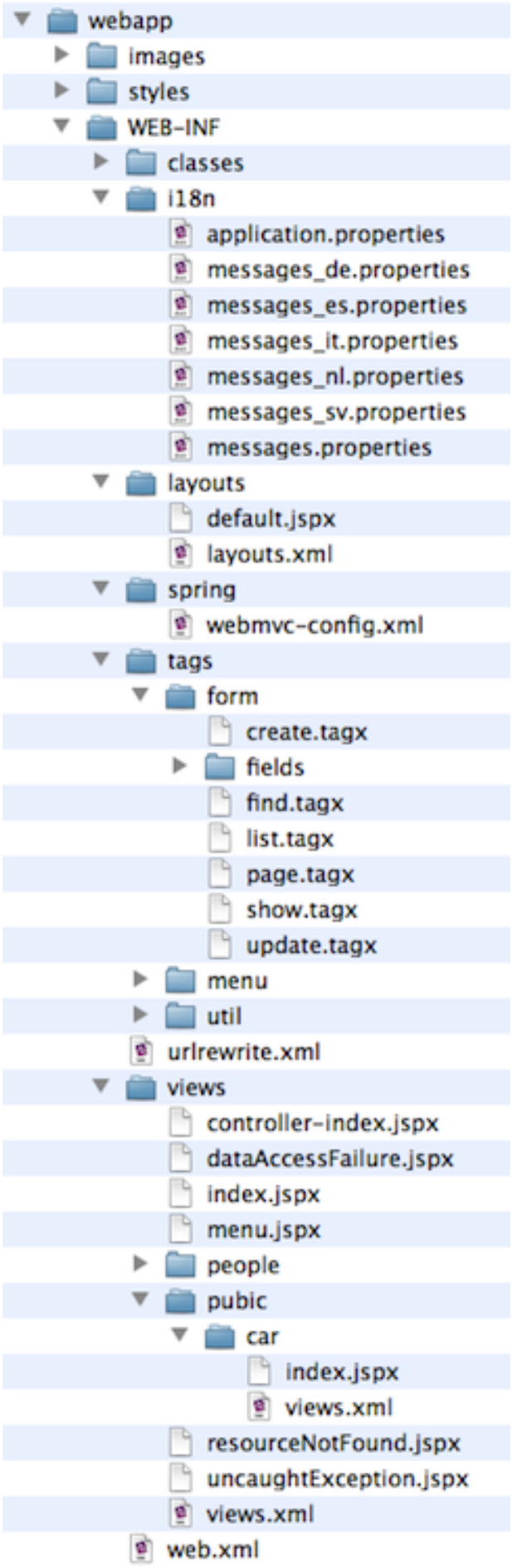
```
@RooConversionService
public class ApplicationConversionServiceFactoryBean extends
FormattingConversionServiceFactoryBean {

    // ...

    public Converter<Vote, String> getVoteConverter() {
        return new Converter<Vote, String>() {
            public String convert(Vote source) {
                return new StringBuilder().append(
                        source.getIp()).append(" ").append(source.getRegistered())
.toString();
            }
        };
    }
}
```

At this point Roo will notice that the addition of the method and will remove it from the ITD much like overriding the toString() method in a Roo entity works.

Note, in some cases you may create a form backing entity which does not contain any suitable fields for conversion. For example, the entity may only contain a field indicating a relationship to another entity (i.e. type one-to-one or one-to-many). Since Roo does not use these fields for its generated converters it will simply omit the creation of a converter for such form backing entities. In these cases you may have to provide your own custom converter to convert from your entity to a suitable String representation in order to prevent potential converter exceptions.

8.3. JSP Views

As mentioned in the previous section, Roo copies a number of static artifacts into the target project after issuing the controller command for the first time. These artifacts include Cascading Style Sheets, images, Tiles layout definitions, JSP files, message property files, a complete tag library and a web.xml file. These artifacts are arranged in different folders which is best illustrated in the following picture:


webapp
images
styles
WEB-INF
classes
i18n
application.properties
messages_de.properties
messages_es.properties
messages_it.properties
messages_nl.properties
messages_sv.properties
messages.properties
layouts
default.jspx
layouts.xml
spring
webmvc-config.xml
tags
form
create.tagx
fields
find.tagx
list.tagx
page.tagx
show.tagx
update.tagx
menu
util
urlrewrite.xml
views
controller-index.jspx
dataAccessFailure.jspx
index.jspx
menu.jspx
people
pubic
car
index.jspx
views.xml
resourceNotFound.jspx
uncaughtException.jspx
views.xml
web.xml

The i18n folder contains translations of the Web UI. The messages_XX.properties files are static resources (which will never be adjusted after the initial installation) which contain commonly used literals which are part of the Web UI. The application.properties file will be managed by Roo to contain application-specific literals. New types or fields added to the domain layer will result in new key/value combinations being added to this file. If you wish to translate the values generated by Roo in the application.properties file, just create a copy of this file and rename it to application_XX.properties (where XX represents your language abbreviation).

Roo uses XML compliant JSP files (JSPX) instead of the more common JSP format to allow round-tripping of views based on changes in the domain layer of your project. Not all jspx files in the target project are managed by Roo after the initial installation (although future addons may choose to do so). Typically jspx files in sub folders under WEB-INF/views are maintained in addition to the menu.jspx.

Here is an example of a typical roo managed jspx file (i.e.: WEB-INF/views/people/update.jspx):

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<div xmlns:field="urn:jsptagdir:/WEB-INF/tags/form/fields"
     xmlns:form="urn:jsptagdir:/WEB-INF/tags/form"
     xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0">
  <jsp:output omit-xml-declaration="yes"/>

  <form:update id="fu_com_foo_Person" modelAttribute="person" path="/people"
               z="3lX+WZW4CQVBb7OlvB0AvdgbGRQ=">
      <field:datetime dateTimePattern="${person_birthday_date_format}" field="birthDay"
                      id="c_com_foo_Person_birthDay" z="dXnEoWaz4rI4CKD9mlz+clbSUP4="/>
      <field:select field="car" id="c_com_foo_Person_car" itemValue="id" items="${cars}"
                    path="/cars" z="z2LA3LvNKRO9OISmZurGjEczHkc="/>
      <field:select field="cars" id="c_com_foo_Person_cars" itemValue="id" items="
${cars}"
                    multiple="true" path="/cars" z="c0rdAISxzHsNvJPFfAmEEGz2LU4="/>
  </form:update>
</div>
```

You will notice that this file is fairly concise compared to a normal jsp file. This is due to the extensive use of the tag library which Roo has installed in your project in the WEB-INF/tags folder. Each tag offeres a number of attributes which can be used to customize the appearance / behaviour of the tag - please use code completion in your favourite editor to review the options or take a peek into the actual tags.

All tags are completely self-reliant to provide their functionality (there are no Java sources needed to implement specific behaviour of any tag). This should make it very easy to customize the behaviour of the default tags without any required knowledge of traditional Java JSP tag development. You are free to customize the contents of the Roo provided tag library to suit your own requirements. You could even offer your customized tag library as a new addon which other Roo users could install to replace the default Roo provided tag library.

Most tags have a few common attributes which adhere with Roo conventions to support round-tripping of the jspx artifacts. The following rules should be considered if you wish to customize tags or jspx files in a Roo managed project:

- The id attribute is used by Roo to find existing elements and also to determine message labels used as part of the tag implementation. Changing a tag identifier will result in another element being generated by Roo when the Roo shell is active.
- Roo provided tags are registered in the root element of the jspx document and are assigned a namespace. You should be able to see element and attribute code completion when using a modern IDE (i.e. SpringSource Tool Suite)
- The z attribute represents a hash key for a given element (see a detailed discussion of the hash key attribute in the paragraph below).

The hash key attribute is important for Roo because it helps determining if a user has altered a Roo managed element. This is the secret to round-trip support for JSPX files, as you can edit anything at any time yet Roo will be able to merge in changes to the JSPX successfully. The hash key shown in the "z" attribute is calculated as shown in the following table:

Included in hash key calculation	*Not* **included in hash key calculation**
Element name (name only, not namespace)	Namespace of element name
Attribute names present in element	White spaces used in the element
Attribute values present in the element	Potential child elements
	The z key and its value
	Any attribute (and value) whose name starts with '_'
	The order of the attributes does not contribute to the value of a hash key

The hash code thus allows Roo to determine if the element is in its "original" Roo form, or if the user has modified it in some way. If a user changes an element, the hash code will not match and this indicates to Roo that the user has customized that specific element. Once Roo has detected such an event, Roo will change the "z" attribute value to "user-managed". This helps clarify to the user that Roo has adopted a "hands off" approach to that element and it's entirely the user's responsibility to maintain. If the user wishes for Roo to take responsibility for the management of a "user-managed" element once again, he or she can simply change the value of "z" to "?". When Roo sees this, it will replace the questionmark character with a calculated hash code. This simple mechanism allows Roo to easily round trip JSPX files without interfering with manual changes performed by the user. It represents a significant enhancement from Roo 1.0 where a file was entirely user managed or entirely Roo managed.

Roo will order fields used in forms in the same sequence they appear in the domain object. The user can freely change the sequence of form elements without interfering with Roo's round tripping approach (Roo will honour user chosen element sequences as long as it can detect individual elements by their id).

The user can nest Roo managed elements in in any structure he wishes without interfering with Roo jspx round tripping. For example elements can be enclosed by HTML div or span tags to change visual or structural appearance of a page.

Most default tags installed by Roo have a render attribute which is of boolean type. This allows users to completely disable the rendering of a given tag (and potential sub tags). This is useful in cases where you don't wish individual fields in a form to be presented to the user but rather have them autopopulated through other means (i.e. input type="hidden"). The value of the render attribute can also be calculated dynamically through the Spring Expression Language (SpEL) or normal JSP expression language. The generated create.jspx in Roo application demonstrates this.

Scaffolding of JPA reference relationships

The Roo JSP addon will read JSR 303 (bean validation API) annotations found in a form-backing object. The following convention is applied for the generation of create and update (and finder) forms:

Data type / JPA annotation	Scaffolded HTML Element
String (sizeMax < 30; @Size)	Input
String (sizeMax >=30, @Size)	Textarea
Number (@Min, @Max, @DecimalMin & @DecimalMax are recognized)	Input
Boolean	Checkbox
Date / Calendar (@Future & @Past are recognized) (Spring's @DateTimeFormat in combination with the *style* or *pattern* attributes is recognized)	Input (with JS Date chooser)
Enum / @Enumerated	Select
@OneToOne	Select
@ManyToMany	Select (multi-select)
@ManyToOne	Select
@OneToMany *	Nothing: A message is displayed explaining that this relationship is managed from the many-side

* As mentioned above, Roo does not scaffold a HTML form element for the 'one' side of a @OneToMany

relationship. To make this relationship work, you need to provide a @ManyToOne annotated field on the opposite side:

```
field set --fieldName students --type com.foo.domain.Person <strong>--class
com.foo.domain.School --cardinality ONE_TO_MANY</strong>

field reference --fieldName school --type com.foo.domain.School --class
com.foo.domain.Person <strong>--cardinality MANY_TO_ONE</strong>
```

In case a field is annotated with @Pattern, the regular expression is passed on to the tag library where it may be applied through the use of the JS framework of choice.

Automatic Scaffolding of dynamic finders

Roo will attempt to scaffold Spring MVC JSP views for all dynamic finders registered in the form backing object. This is done by using the web mvc finder all or web mvc finder add command.

Due to file name length restrictions by many file systems (see http://en.wikipedia.org/wiki/Comparison_of_file_systems) Roo can only generate JSP views for finders which have 244 characters or less (including folders). If the finder name is longer than 244 characters Roo will silently skip the generation of jsp view artifacts for the dynamic finder in question). More detail can be found in ticket ROO-1027.

Chapter 9. Cloud Add-On

Spring Roo provides integration with Cloud including Cloud providers in the generated project.

One of these providers is Cloud Froundry, that includes a maven plugin in your generated project.

To add cloud foundry support in the generated project is necessary to execute the following command:

```
roo> cloud setup --provider CLOUD_FOUNDRY
Congratulations! Now you can use Cloud Foundry Maven Plugin to deploy your applications!

WARNING: You don't specify any configuration.

You can use the following Cloud Foundry Maven Plugin commands with "perform command
--mavenCommand" on ROO Shell or using "mvn" on OS command line.

Command                        Description
---------------------------------------
cf:apps                         List deployed applications.
cf:app                           Show details of an application.
cf:delete                       Delete an application.
cf:env                           Show an application's environment variables.
cf:help                        Show documentation for all available commands.
cf:push                        Push and optionally start an application.
cf:push-only                   Push and optionally start an application, without
packaging.
cf:restart                       Restart an application.
cf:start                       Start an application.
cf:stop                       Stop an application.
cf:target                      Show information about the target Cloud Foundry service.
cf:logs                      Tail application logs.
cf:recentLogs                Show recent application logs.
cf:scale                      Scale the application instances up or down.
cf:services                  Show a list of provisioned services.
cf:service-plans              Show a list of available service plans.
cf:create-services              Create services defined in the pom.
cf:delete-services              Delete services defined in the pom.
cf:bind-services              Bind services to an application.
cf:unbind-services              Unbind services from an application.
cf:delete-orphaned-routes     Delete all routes that are not bound to any application.
cf:login                      Log in to the target Cloud Foundry service and save access
tokens.
cf:logout                      Log out of the target Cloud Foundry service and remove
access tokens.
Updated ROOT/pom.xml [added plugin org.cloudfoundry:cf-maven-plugin:1.0.4]
```

After that, follow the instructions and configure your project to be deployed on Cloud Foundry.

Chapter 10. JSON Add-On

There are a number of ways to work with JSON document serialization and desrialization in Roo projects:

Option 1: Built-in JSON handling managed in domain layer (discussed in this section)

- This offers customizable FlexJson integration

Option 2: Spring MVC detects the Jackson library in the application classpath

- simply use Spring's @RequestBody and @ResponseBody annotations in the controllers, or
- take advantage of Spring's ContentNegotiatingViewResolver

The JSON add-on offers JSON support in the domain layer as well as the Spring MVC scaffolding. A number of methods are provided to facilitate serialization and deserialization of JSON documents into domain objects. The JSON add-on makes use of the Flexjson library.

10.1. Adding JSON Functionality to Domain Types

The add-on offers an annotation as well as two commands that support the integration of JSON support into the project's domain layer:

1. Annotating a target type with the default **@RooJson** annotation will prompt Roo to create an ITD with the following four methods:

```
public String toJson() {
   return new JSONSerializer().exclude("*.class").serialize(this);
}
```

   This method returns a JSON representation of the current object.

```
public static Owner fromJsonToOwner(String json) {
   return new JSONDeserializer<Owner>().use(null, Owner.class).deserialize(json);
}
```

   This method has a String parameter representing the JSON document and returns a domain type instance if the document can be serialized by the underlying deserializer.

```
public static String toJsonArray(Collection<Owner> collection) {
   return new JSONSerializer().exclude("*.class").serialize(collection);
}
```

This method will convert a collection of the target type, provided as method parameter, into a valid JSON document containing an array.

```
public static Collection<Owner> fromJsonArrayToOwners(String json) {
   return new JSONDeserializer<List<Owner>>().use(null,
                     ArrayList.class).use("values", Owner.class).deserialize(json);
}
```

This method will convert a JSON array document, passed in as a method parameter, into a collection of the target type.

The @RooJson annotation can be used to customize the names of the methods being introduced to the target type. Furthermore, you can disable the creation of any of the above listed methods by providing an empty String argument for the unwanted method in the @RooJson annotation. Example:

```
@RooJson(toJsonMethod="", fromJsonMethod="myOwnMethodName")
```

2. The json add Roo shell command will introduce the @RooJson annotation into the specified target type.

3. The json all command will detect all domain entities in the project and annotate all of them with the @RooJson annotation.

10.2. JSON REST Interface in Spring MVC controllers

Once your domain types are annotated with the @RooJson annotation, you can create Spring MVC scaffolding for your JSON enabled types.

1. The web mvc json setup Roo shell command configures the current project to support JSON integration using Spring MVC.

2. The web mvc json add Roo shell command introduces the **@RooWebJson** annotation into the specified target type.

3. The web mvc json all Roo shell command finds all JSON-enabled types (**@RooJson**) in the project and creates Spring MVC controllers for each (if a controller does not already exist), or adds **@RooWebJson** to existing controllers (should they already exist).

4. Annotating an existing Spring MVC controller with the @RooWebJson annotation will prompt Roo to create an ITD with a number of methods:

   - listJson

   ```
   @RequestMapping(headers = "Accept=application/json")
   @ResponseBody
   public ResponseEntity<String> ToppingController.listJson() {
       HttpHeaders headers = new HttpHeaders();
       headers.add("Content-Type", "application/json; charset=utf-8");
       List<Topping> result = toppingService.findAllToppings();
       return new ResponseEntity<String>(Topping.toJsonArray(result), headers,
   HttpStatus.OK);
   }
   ```

   As you can see this method takes advantage of Spring's request mappings and will respond to HTTP GET requests that contain an 'Accept=application/json' header. The @ResponseBody annotation is used to serialize the JSON document.

   To test the functionality with curl, you can try out the Roo "pizza shop" sample script (run roo> script pizzashop.roo; then quit the Roo shell and start Tomcat 'mvn tomcat:run'):

   ```
   curl -i -H "Accept: application/json" http://localhost:8080/pizzashop/toppings
   ```

   - showJson

   ```
   @RequestMapping(value = "/{id}", headers = "Accept=application/json")
   @ResponseBody
   public ResponseEntity<String> ToppingController.showJson(@PathVariable("id") Long
   id) {
       Topping topping = toppingService.findTopping(id);
       HttpHeaders headers = new HttpHeaders();
       headers.add("Content-Type", "application/json; charset=utf-8");
       if (topping == null) {
           return new ResponseEntity<String>(headers, HttpStatus.NOT_FOUND);
       }
       return new ResponseEntity<String>(topping.toJson(), headers, HttpStatus.OK);
   }
   ```

   This method accepts an HTTP GET request with a @PathVariable for the requested Topping ID. The entity is serialized and returned as a JSON document if found, otherwise an HTTP 404 (NOT FOUND) status code is returned. The accompanying curl command is as follows:

```
curl -i -H "Accept: application/json" http://localhost:8080/pizzashop/toppings/1
```

- createFromJson

```
@RequestMapping(method = RequestMethod.POST, headers = "Accept=application/json")
public ResponseEntity<String> ToppingController.createFromJson(@RequestBody String
json) {
    Topping topping = Topping.fromJsonToTopping(json);
    toppingService.saveTopping(topping);
    HttpHeaders headers = new HttpHeaders();
    headers.add("Content-Type", "application/json");
    return new ResponseEntity<String>(headers, HttpStatus.CREATED);
}
```

This method accepts a JSON document sent via HTTP POST, converts it into a Topping instance, persists that new instance, and returns an HTTP 201 (CREATED) status code. The accompanying curl command is as follows:

```
curl -i -X POST -H "Content-Type: application/json" -H "Accept: application/json"
     -d '{"name": "Thin Crust"}' http://localhost:8080/pizzashop/bases
```

- createFromJsonArray

```
@RequestMapping(value = "/jsonArray", method = RequestMethod.POST, headers =
"Accept=application/json")
public ResponseEntity<String> ToppingController.createFromJsonArray(@RequestBody
String json) {
    for (Topping topping: Topping.fromJsonArrayToToppings(json)) {
        toppingService.saveTopping(topping);
    }
    HttpHeaders headers = new HttpHeaders();
    headers.add("Content-Type", "application/json");
    return new ResponseEntity<String>(headers, HttpStatus.CREATED);
}
```

This method accepts a document containing a JSON array sent via HTTP POST and converts the array into instances that are then persisted. The method returns an HTTP 201 (CREATED) status code. The accompanying curl command is as follows:

```
curl -i -X POST -H "Content-Type: application/json" -H "Accept: application/json"
     -d '[{"name":"Cheesy Crust"},{"name":"Thick Crust"}]'
     http://localhost:8080/pizzashop/bases/jsonArray
```

- updateFromJson

```
@RequestMapping(method = RequestMethod.PUT, headers = "Accept=application/json")
public ResponseEntity<String> ToppingController.updateFromJson(@RequestBody String
json) {
    HttpHeaders headers = new HttpHeaders();
    headers.add("Content-Type", "application/json");
    Topping topping = Topping.fromJsonToTopping(json);
    if (toppingService.updateTopping(topping) == null) {
        return new ResponseEntity<String>(headers, HttpStatus.NOT_FOUND);
    }
    return new ResponseEntity<String>(headers, HttpStatus.OK);
}
```

This method accepts a JSON document sent via HTTP PUT and converts it into a Topping instance before attempting to merge it with an existing record. If no existing record is found, an HTTP 404 (NOT FOUND) status code is sent to the client, otherwise an HTTP 200 (OK) status code is sent. The accompanying curl command is as follows:

```
curl -i -X PUT -H "Content-Type: application/json" -H "Accept: application/json"
     -d '{id:6,name:"Mozzarella",version:1}'
     http://localhost:8080/pizzashop/toppings
```

- updateFromJsonArray

```
@RequestMapping(value = "/jsonArray", method = RequestMethod.PUT,
                headers = "Accept=application/json")
public ResponseEntity<String> BaseController.updateFromJsonArray(@RequestBody String
json) {
    HttpHeaders headers = new HttpHeaders();
    headers.add("Content-Type", "application/json");
    for (Base base: Base.fromJsonArrayToBases(json)) {
        if (baseService.updateBase(base) == null) {
            return new ResponseEntity<String>(headers, HttpStatus.NOT_FOUND);
        }
    }
    return new ResponseEntity<String>(headers, HttpStatus.OK);
}
```

This method accepts a document containing a JSON array sent via HTTP PUT and converts the array into transient entities which are then merged. The method returns an HTTP 404 (NOT FOUND) status code if any of the instances to be updated are not found, otherwise it returns an HTTP 200 (OK) status code. The accompanying curl command is as follows:

```
curl -i -X PUT -H "Content-Type: application/json" -H "Accept: application/json"
     -d '[{id:1,"name":"Cheesy Crust",version:0},{id:2,"name":"Thick
Crust",version:0}]'
     http://localhost:8080/pizzashop/bases/jsonArray
```

- deleteFromJson

```
@RequestMapping(value = "/{id}", method = RequestMethod.DELETE, headers =
"Accept=application/json")
public ResponseEntity<String> ToppingController.deleteFromJson(@PathVariable("id")
Long id) {
    Topping topping = toppingService.findTopping(id);
    HttpHeaders headers = new HttpHeaders();
    headers.add("Content-Type", "application/json");
    if (topping == null) {
        return new ResponseEntity<String>(headers, HttpStatus.NOT_FOUND);
    }
    toppingService.deleteTopping(topping);
    return new ResponseEntity<String>(headers, HttpStatus.OK);
}
```

This method accepts an HTTP DELETE request with an @PathVariable identifying the Topping instance to be deleted. HTTP status code 200 (OK) is returned if a Topping with that ID was found, otherwise HTTP status code 404 (NOT FOUND) is returned. The accompanying curl command is as follows:

```
curl -i -X DELETE -H "Accept: application/json"
http://localhost:8080/pizzashop/toppings/1
```

- jsonFind...

[Optional] Roo will also generate a method to retrieve a document containing a JSON array if the form backing object defines dynamic finders. Here is an example taken from VisitController in the pet clinic sample application, after adding JSON support to it:

```
@RequestMapping(params = "find=ByDescriptionAndVisitDate", method = RequestMethod
.GET,
                headers = "Accept=application/json")
public String jsonFindVisitsByDescriptionAndVisitDate(@RequestParam("description")
String desc,
        @RequestParam("visitDate") @DateTimeFormat(style = "M-") Date visitDate,
Model model) {
    return Visit.toJsonArray(Visit.findVisitsByDescriptionAndVisitDate(desc,
visitDate).getResultList());
}
```

This method accepts an HTTP GET request with a number of request parameters which define the finder method as well as the finder method arguments. The accompanying curl command is as follows:

```
curl -i -H Accept:application/json
     http://localhost:8080/petclinic/visits?find=ByDescriptionAndVisitDate%26descrip
tion=test%26visitDate=12/1/10
```

If you need help configuring how FlexJson serializes or deserializes JSON documents, please refer to their reference documentation.

Chapter 11. Community Add-Ons

Besides add-ons distributed with Spring Roo, exists other add-ons maintained by the community that in the past were part of Spring Roo distribution. We call them "Community Add-Ons".

These add-ons were developed by Spring Roo team, officially endorsed to community partners and no longer maintained by Spring Roo team. The release cycle of these addons is separated from the normal Roo repository so that each addon in the Spring Roo Community repository can be released as part of its own release cycle.

The Spring Roo Community addons are NOT released during the normal Roo release cycle and the Roo CI server does NOT perform any addon releases.

If you are interested on community add-ons, you can visit Spring Roo Community Addons gitHub repository.

Chapter 12. Manage Roo Add-Ons

It is easy to extend the capabilities of Spring Roo with installable add-ons. This section will offer a basic overview of Roo's add-on distribution model and explain how to install new add-ons. If you're considering writing an add-on, please refer to the more advanced information in Part III of this reference guide.

First of all, it's important to recognize that Roo ships with a large number of base add-ons. These built-in add-ons may be all you ever require. Nevertheless, there is a growing community of add-ons written by people outside the core Roo team. Because the core Roo team do not write these add-ons, we've needed to implement an infrastructure so that external people can share their add-ons and make it easy for you to install them.

Roo's add-on distribution system encourages individual add-on developers to host their add-on web site (we don't believe in a central model where we must host add-ons on our servers). The main requirement an add-on developer needs to fulfill is their add-ons must be in OSGi format and their web site must include an OSGi Bundle Repository (OBR) index file. While Roo internally uses OSGi and all modules are managed as OSGi bundles, this is transparent and you do not need any familiarity with OSGi or bundles to work with the Roo add-on installation system. An OBR file is usually named `index.xml` and it is available over HTTP. If you're curious what these OBR files look like, you can view the Spring Roo OBR repository at http://repo.spring.io/spring-roo/index.xml. Within an OBR file each available Roo-related add-on is listed, along with the URL where it is published. The URLs look similar to normal URLs.

12.1. OSGi Repositories

An OSGi Repository is a collection of OSGi bundles. Optionally, the repository contains an index file which reports the content of the repository along with the capabilities and requirements of each resource listed.

A repository may be located anywhere: somewhere else on the local file system; or on a remote server.

The OSGi repositories provides to Roo the capability to search, retrieve and install addons from remote servers.

Spring Roo includes a default OSGi repository (http://repo.spring.io/spring-roo/index.xml) that contains external components that could be used by developers during development.

To show all installed OSGi repositories in your Spring Roo distribution, execute:

```
roo> addon repository list
Getting current installed repositories...

   http://repo.spring.io/spring-roo/index.xml

1 installed repositories on Spring Roo were found
```

If you are interested on install new OSGi repositories on your Spring Roo distrbution you can execute the following commands:

```
roo> addon repository add --url http://localhost/repo/index.xml
```

NOTE | *http://localhost/repo/index.xml is the index file URL of the repository you want install.*

To remove some installed OSGi repository you can do it using addon repository remove command:

```
roo> addon repository remove --url http://localhost/repo/index.xml
```

Spring Roo provides you commands to locate new available add-ons on installed OSGi repositories. To get the list of known add-ons you can use the addon repository introspect or addon search command.

For example, addon repository introspect lists all add-ons that are in the installed OSGi repositories mentioned above:

```
roo> addon repository introspect
Getting available bundles on installed repositories...

Status               Bundle Description and version
-------------------------------------------------------------------------------
Not Installed        Spring Roo - Wrapping - derby (10.8.2.2_0002)
Not Installed        Spring Roo - Wrapping - derbyclient (10.8.2.2_0002)
Not Installed        Spring Roo - Wrapping - firebird (2.1.6.0021)
Not Installed        Spring Roo - Wrapping - jtds (1.2.4.0011)
Not Installed        Spring Roo - Wrapping - jtopen (6.7.0.0011)
Not Installed        Spring Roo - Wrapping - mysql-connector-java (5.1.18.0002)
Not Installed        Spring Roo - Wrapping - postgresql-jdbc3 (9.1.0.901_0003)

7 available bundles on installed repositories were found
```

12.2. Available Add-Ons in OSGi Repositories

To review details about a specific add-on, use the addon info bundle which requires a "bundle

symbolic name", which is usually the add-on's top-level package. Let's try this. To view details about the second add-on listed above, enter this command:

```
roo> addon info bundle --bundleSymbolicName org.springframework.roo.wrapping.derbyclient
```

An example of the output of addon info bundle is shown below:

```
addon info bundle --bundleSymbolicName org.springframework.roo.wrapping.derbyclient
Name.........: Spring Roo - Wrapping - derbyclient
BSN..........: org.springframework.roo.wrapping.derbyclient
Version......: 10.8.2.2_0002
JAR Size.....: 547922 bytes
JAR URL......: http://repo.spring.io/spring-roo/org/springframework/roo/wrapping
               /org.springframework.roo.wrapping.derbyclient/10.8.2.2.0002/org.s
               pringframework.roo.wrapping.derbyclient-10.8.2.2.0002.jar
```

In the above output "BSN" means bundle symbolic name, which is the alternate way of referring to a given add-on. The output also shows you the Roo shell commands that are available via the add-on if have it. These commands are automatically seen by the Roo shell, so if you typed the command without first having installed the add-on, Roo would have performed a search and shown you this add-on offered the command. This is a great feature and means you can often just type commands you think you might need and find out which add-ons offer them without performing an explicit search. A similar feature exists for JDBC resolution if you try to reverse engineer a database for which there is no installed JDBC driver (Roo will automatically suggest the add-on you need and instruct you which command to use to install it).

If you decide to install a specific add-on, simply use the addon install bundle command:

```
roo> addon install bundle --bundleSymbolicName
org.springframework.roo.wrapping.postgresql-jdbc3
Target resource(s):
-------------------
   Spring Roo - Wrapping - postgresql-jdbc3 (9.1.0.901_0003)

Deploying...
done.

Starting org.springframework.roo.wrapping.postgresql-jdbc3; id: 86 ...
Started!

Successfully installed add-on: Spring Roo - Wrapping - postgresql-jdbc3 [version:
9.1.0.901_0003]
```

Of course, you can remove add-ons as well. To uninstall any given add-on, just use the addon remove command. On this occasion we'll use the bundle symbolic name (which is available via TAB completion as is usual with Roo):

```
roo> addon remove --bundleSymbolicName org.springframework.roo.wrapping.postgresql-jdbc3
Bundle 'org.springframework.roo.wrapping.postgresql-jdbc3' : Uninstalled!
```

12.3. Available Roo Add-On Suites in OSGi Repositories

A "Roo Addon Suite" is a great way to package and distribute a set of add-ons together, for example if you want to distribute Roo custom distributions.

Roo Addon Suite is based on OSGi R5 Subsystems that provides a really convenient deployment model, without compromising the modularity of Roo.

To know which Roo Add-On Suites are installed on your Spring Roo distribution, you could execute the following command:

```
roo> addon suite list
```

All Roo Add-On Suites should be located in OSGi Repositories, so to install a new Roo Add-On Suite on your Spring Roo distribution is necessary to install an OSGi Repository as we saw above.

After install the OSGi repository that contains the Roo Add-On Suite, you could install all available Roo Add-On Suites executing the following command:

```
roo> addon suite install name --symbolicName org.mysuite.roo.addon.suite
```

NOTE *Symbolic Name* param will be autocompleted with available Roo Add-On Suites on installed OSGi Repositories.

After install the selected Roo Add-On Suite, you will have available on your Spring Roo distribution all components of that Suite. If you want to stop them, you can do it executing the following command:

```
roo> addon suite stop --symbolicName org.mysuite.roo.addon.suite
```

NOTE *Visit Spring Roo Marketplace to get available Roo Add-On Suites repositories.*

12.4. Conclusions

Note that all of the "addon" commands only work with add-ons listed in the installed OSGi Repositories.

If you want to manage your installed OSGi Repositories, see OSGi Repositories section

III. Add-On Development

In this part of the guide we reveal how Roo works internally. With this knowledge you'll be well-positioned to be able to check out the Roo codebase, build a development release, and write add-ons to extend Roo.

You should be familiar with Part I of this reference guide and ideally have used Roo for a period of time to gain the most value from this part.

Chapter 13. Development Processes

In this chapter we'll cover how we develop Roo, and how you can check it out and get involved.

13.1. Guidelines We Follow

Whether you are part of the Roo core development team, you want to contribute patches, or you want to develop add-ons there are a few guidelines we would like to bring to your attention.

1. Design Goals
    - High productivity for Java developers
        - Encourage reuse of existing knowledge, skills and experience
    - Eliminate barriers to adoption, no runtime component, minimal size, best possible development experience
        - Avoid lock-in
        - No runtime component
        - Minimal download size
        - Best possible development experience
    - Embrace the strengths of Java
        - Development-time: tooling, popularity, API quality, static typing
        - Deploy-time: performance, memory use, footprint
2. Embrace the advantages of AspectJ
    - Use AspectJ inter-type declarations (ITDs) for "active" generation
        - Active generation automatically maintains output
    - Delivers compilation unit separation of concerns
        - Easier for users, and easier for us as developers
    - Instant IDE support
        - Reduce time to market and adoption barriers
    - Other good reasons

    - Mature, “push in” refactor, compile-time is welcome

3. ITD Model
    - Roo owns *_Roo_*.aj files
        - Will delete them if necessary
    - Every ITD-providing add-on registers a 'suffix' (namespace)
        - E.g. the 'Entity' add-on provides *_ROO_JPA_ACTIVE_RECORD.aj
        - A missing ITD provider causes AJ file removal
    - ITDs have proper import management
        - So they look and feel normal to developers
        - So they 'push-in refactor' in a natural form
4. Usability = Highest Priority
    - Interactivity of Roo Shell
    - Tab completion, context awareness, command hiding, hint support, etc
    - Background monitoring of externally made changes (allows integration with any development style)
    - Background monitoring to avoid crude 'generation' steps
5. Immutability of Metadata Types
    - Immutability as a first step to manage concurrency
    - String-based keys (start with 'MID:')
    - Metadata and keys built on demand only (never persisted)
    - Metadata can depend on other metadata
        - if 'upstream' metadata changes, 'downstream' metadata is notified
        - Some metadata will want to monitor the file system
    - Central metadata service available and cache is provided to enhance performance
6. Conventions we follow
    - Ensure usability is first-class

- Minimize the JAR footprint that Roo requires
- Relocate runtime needs to sister Spring projects
- Embrace immutability as much as possible
- Maximize performance in generated code
- Minimize memory consumption in generated code
- Use long artifact IDs to facilitate identification
- Don't put into @Roo* what you could calculate
- Don't violate generator predictability conventions

13.2. Source Repository

We develop against a public Git repository from which you can anonymously checkout the code:

```
git clone https://github.com/spring-projects/spring-roo.git spring-roo
```

Roo itself uses Maven, so it's very easy to build the standard package, install, assembly and site goals. PGP should be installed, see the 'Setting Up for Development' section below for details.

13.3. Setting Up for Development

We maintain up-to-date documentation in the `readme.txt` in the root of the checkout location. Please follow these instructions carefully.

13.4. Submitting Patches

Submitting a patch for a bug, improvement or even a new feature which you always wanted addressed can be of great help to the Spring Roo project.

To get started, create new ticket on our JIRA to warn the community with your new feature, improvement or bug fix.

You could build Roo from sources (as described above), and locally start changing source code as you see fit. Then test your changes and if all works well, you can create a git patch and attach it to the JIRA ticket.

Feel free to submit a pull request at https://github.com/spring-projects/spring-roo/pulls to fix this issue.

13.5. Path to Committer Status

Essentially if you submit a patch and we think it is useful to commit to the code base, we will ask you to complete our contributor agreement. This is just a simple web form that deals with issues like patents and copyrights. Once this is done, we can apply your patch to the source code repository.

Feel free to complete the online individual contributers agreement at https://support.springsource.com/spring_committer_signup so we can apply your patch.

Chapter 14. Simple Add-Ons

This chapter will provide an introduction to Spring Roo add-on development. The intention is to provide a step-by-step guide that walks the developer from zero code to a fully deployed and published add-on that is immediately available to all Spring Roo users.

A new add-on named 'Add-On Creator' has been developed that facilitates the creation of a new Spring Roo simple add-ons, advanced add-ons and roo add-on suites.

The following sections will present a complete step-by-step guide demonstrating how to bootstrap a new Spring Roo add-on, publish, release it and register it on Spring Roo Marketplace

OSGi in Spring Roo

Spring Roo runs in an OSGi container since version 1.1. This internal change is ideal for Roo's add-on model because it allows Roo users to install, uninstall, start, and stop diferent add-ons dynamically without restarting the Roo shell. Furthermore, OSGi allows automatic provisioning of external add-on repositories and provides very good infrastructure for developing modular, as well as embedded, and service-oriented applications. Under the hood, Spring Roo uses the Apache Felix OSGi implementation.

Spring Roo runs in an OSGi R5 container since version 2.0. This improvement allows Roo to include new OSGi R5 services like Subsystem Service, Repository Service, Config Administration and HTTP Service among others. These OSGi services give to Roo the ability to implement new features for the developers like the Roo Addon Suites and the Web UI.

14.1. Fast Creation

Roo's Add-On Creator Commands

Spring Roo offers the following commands to help developers quickly create new add-ons:

- **addon create simple**
  - *What*: Command & Operations support
  - *When*: Simple add-ons that want to add dependencies and/or configuration artifacts to a project
- **addon create advanced**
  - *What*: Command, Operations & ITD support
  - *When*: Full-fledged add-ons that offer new functionality to project enhancements to existing Java types in project introduction of new Java types (+ ITDs)
- **addon create i18n**
  - *What*: Extension to the existing 'web mvc install language' command
  - *When*: A new translation is added to the Spring MVC admin UI scaffolding
- **addon create wrapper**
  - *What*: Wrapping of a Maven artifact with an OSGi compliant manifest
  - *When*: A dependency is needed to complete other functionality offered by a Roo add-on (for example a JDBC driver for the DBRE add-on)

Once you have installed Java, Maven, PGP you can change into the <project-name> directory. In the <project-name> directory, you can start the Spring Roo shell and use one of the new commands for add-on creation:

```
roo> addon create simple --topLevelPackage org.simple.addon --projectName "Simple Addon"
```

The addon create simple command will scaffold a number of artefacts:

```
roo> addon create simple --topLevelPackage org.simple.addon --projectName "Simple Addon"
Created ROOT/pom.xml
Created ROOT/readme.txt
Created ROOT/legal
Created ROOT/legal/LICENSE.TXT
Created SRC_MAIN_JAVA/org/simple/addon
Created SRC_MAIN_JAVA/org/simple/addon/AddonCommands.java
Created SRC_MAIN_JAVA/org/simple/addon/AddonOperations.java
Created SRC_MAIN_JAVA/org/simple/addon/AddonOperationsImpl.java
Created SRC_MAIN_JAVA/org/simple/addon/AddonPropertyName.java
Created SRC_MAIN_RESOURCES/org/simple/addon
Created SRC_MAIN_RESOURCES/org/simple/addon/info.tagx
Created SRC_MAIN_RESOURCES/org/simple/addon/show.tagx
Updated SRC_MAIN_RESOURCES/obr.xml
```

This newly created add-on project can be imported into the STS via File > Import > Maven > Existing Maven projects. Let's discuss some of these artefacts in more detail:

1. **pom.xml** - This is the Maven project configuration. This configuration ships with a number of preinstalled Maven plugins that facilitate the PGP artefact signing process as well as the project release process (including tagging etc). It also adds the OSGi and Felix dependencies needed for the addon to run in the Roo Shell. Furthermore, several commonly used Spring Roo modules are preinstalled. These modules provide functionalities such as file system monitoring, Roo shell command registration, etc. More information about these functionalities is provided in the following sections.

   The add-on developer should open up the pom.xml file and modify some project specific references and documentation (marked in bold font):

   ```
   <?xml version="1.0" encoding="UTF-8" standalone="no"?>
   <project [...]>
     [...]
     <name>com-foo-batch</name>
     <organization>
       <name>Your project/company name goes here (used in copyright and vendor information
   in the manifest)</name>
     </organization>
     [...]
     <description>An add-on created by Spring Roo's addon creator feature.</description>
     <url>http://www.some.company</url>
     <properties>
   ```

   Some of these properties can also be provided when issuing the addon create command.

2. **readme.txt** - You can provide any setup or installation information about your add-on in this file.

This file is used by other developers who checkout your add-on source code from the SVN repository.

3. **legal/LICENSE.TXT** - Copy the appropriate license text for your add-on into this file.

4. **src/main/java/com/foo/batch/BatchCommands.java** - This is a fully working code example demonstrating how to register commands offered by your addon into the Spring Roo Shell (more detailed information in the next section).

5. **src/main/java/com/foo/batch/BatchOperations.java & BatchOperationsImpl.java** - These artefacts are used to perform operations triggered by a command (more information in the next sections).

6. **src/main/java/com/foo/batch/BatchPropertyName.java** - This type provides a simple example demonstrating the use of static command completion options for the Spring Roo Shell. An example of static command completion options are for example the database selection options as part of the jpa setup command.

7. **src/main/assembly/assembly.xml** - This artefact defines configurations used for the packaging of the add-on.

14.2. Shell Interaction

Spring Roo provides an easy way for external add-ons to contribute new commands to the Roo Shell. Looking at the code extract below, there are really only two artefacts needed in your command type to register a new command in the Roo Shell; your type needs to implement the **CommandMarker** interface, and you need to create a method annotated with **@CliCommand**. Let us review some details:

```
[1] @Component
[1] @Service
[2] public class BatchCommands implements CommandMarker {

[3] @Reference private BatchOperations operations;
    @Reference private StaticFieldConverter staticFieldConverter;

[4] protected void activate(ComponentContext context) {
       staticFieldConverter.add(BatchPropertyName.class);
    }

[5] protected void deactivate(ComponentContext context) {
       staticFieldConverter.remove(BatchPropertyName.class);
    }

[6] @CliAvailabilityIndicator("welcome property")
    public boolean isPropertyAvailable() {
       return operations.isProjectAvailable();
    }

[7] @CliCommand(value="welcome property", help="Obtains a pre-defined system property")
    public String property(@CliOption(key="name", mandatory=false, specifiedDefaultValue
="USERNAME", unspecifiedDefaultValue="USERNAME", help="The property name you'd like to
display") BatchPropertyName propertyName) {
      return operations.getProperty(propertyName);
    }
```

There are a few artefacts of interest when developing Spring Roo add-ons:

1. To register components and services in the Roo shell, the type needs to be annotated with the **@Component** & **@Service** annotations provided by Felix. These components can be injected into other add-ons (more interesting for functionalities exposed by operations types).

2. The command type needs to implement the **CommandMarker** interface, which Spring Roo scans for in order to detect classes that contribute commands to the Roo Shell.

3. The Felix **@Reference** annotations are used to inject services and components offered by other Spring Roo core components or even other add-ons. In this example, we are injecting a reference to the add-on's own BatchOperations interface and the StaticFieldConverter component offered by the Roo Shell OSGi bundle. The Felix **@Reference** annotation is similar in purpose to Spring's @Autowired and @Inject annotations.

4. The **activate** and **deactivate** methods can optionally be implemented to get access to the lifecycle of the addon's bundle as managed by the underlying OSGi container. Roo add-on developers can use these lifecycle hooks for registration and deregistration of converters (typically in command types) or for the registration of metadata dependencies (typically in ITD-providing add-ons) or any other

component initialization activities.

5. The optional **@CliAvailabilityIndicator** annotation allows you to limit when a command is available in the Spring Roo Shell. Methods thus annotated should return a boolean to indicate whether a command should be visible to the Roo Shell. For example, many commands are hidden before a project has been created.

6. The **@CliCommand** annotation plays a central role for Roo add-on developers. It allows the registration of new commands for the Roo Shell. Methods annotated with **@CliCommand** can optionally return a String value to contribute a log statement to the Spring Roo Shell. Another, more flexible, option to provide log statements in the Roo Shell is to register a standard JDK logger, which allows the developer to present color-coded messages to the user in the Roo shell, with the color coding being dependent on the log level (warning, info, error, etc).

7. The optional **@CliOption** annotation can be used to annotate method parameters. These parameters define command attributes that are presented as part of a command. Roo will attempt to automatically convert user-entered values into the Java type of the annotated method parameter. In the example above, Roo will convert the user-entered String to a BatchPropertyName. By default, Roo offers converters for common number types, String, Date, Enum, Locale, boolean and Character. See the *org.springframework.roo.shell.converters* package for examples if you need to implement a custom converter.

14.3. Operations

Almost all Spring Roo add-ons provide operations types. These types do most of the work behind Roo's passive generation principle (active generation is taken care of by AspectJ Intertype declarations (ITDs) - more about that later). Methods offered by the operations types provided by the add-on are typically invoked by the accompanying "command" type. Alternatively, operations types can also be invoked by other add-ons (this is a rather unusual case).

Implementations of the Operations interface need to be annotated with the Felix **@Component** and **@Service** annotations to make their functionality available within Roo's OSGi container. Dependencies can be injected into operations types via the Felix **@Reference** annotation. If the dependency exists in a package that is not yet registered in the add-on's pom.xml, you need to add the dependency there to add the relevant bundle to the add-on's classpath.

The Add-On Creator generated project includes example code which uses Roo's source path abstractions, file manager and various Util classes that take care of project file management.

Typical functionality offered by operations types include:

- Adding new dependencies, plugins, & repositories to the Maven project pom.xml.
- Copying static artefacts from the add-on jar into the user project (i.e. CSS, images, tagx, configuration files, etc).

- Configuring application contexts, web.xml, and other config artefacts.
- Managing properties files in the user project.
- Creating new Java source types in the user project.
- Adding trigger (or other) annotations to target types (most common), fields or methods.

Spring Roo offers a wide range of abstractions and metadata types that support these use cases. For example, the following services are offered:

- org.springframework.roo.process.manager.**FileManager**
  - use file manager for all file system operations in project (offers automatic undo on exception)
- org.springframework.roo.project.**PathResolver**
  - offers abstraction over common project paths
- org.springframework.roo.metadata.**MetadataService**
  - offers access to Roo metadata bean info metadata for mutators/accessors of target type
- org.springframework.roo.project.**ProjectMetadata**
  - project name, top level package read access to project dependencies, repositories, etc
- org.springframework.roo.project.**ProjectOperations**
  - add, remove project Maven dependencies, plugins, repositories, filters, properties, etc

In addition the org.springframework.roo.support bundle provides a number of useful utils classes:

- org.springframework.roo.support.util.**Assert**
  - similar to Spring's Assert, exceptions thrown by Assert will cause Roo's File manager abstraction to roll back.
- org.springframework.roo.support.util.**FileCopyUtils**
  - useful for copying resources from add-on into project
- org.springframework.roo.support.util.**TemplateUtils**
  - useful for obtaining InputStream of resources in bundle
- org.springframework.roo.support.util.**XmlUtils**
  - hides XML ugliness

- writeXml methods
- Xpath abstraction & cache
- XML Transformer setup

14.4. Packaging & Distribution

Once your add-on is complete, you can test its functionality locally by generating an OSGi-compliant jar bundle and installing it in the Spring Roo Shell:

```
<project-name>$ <strong>mvn clean install</strong>
```

This will generate your add-on OSGi bundle in the project's *target* directory. In a separate directory, you can start the Spring Roo Shell and use the following command to test your new add-on:

```
roo> <strong>addon install url</strong> --url file:///<path-to-addon-
project/target/<addon-bundle-name>.<version>.jar
```

This should install and activate your new Spring Roo Add-On. For troubleshooting, Roo offers the following OSGi commands:

- **addon list**- Displays OSGi bundle information & status. This should list your add-on as active.

Once you have tested the add-on successfully in your development environment, you can release the add-on jar to your own OSGi Repository and update the Spring Roo Marketplace with yout OSGi Repository URL.

Chapter 15. Roo Addon Suites

A “Roo Addon Suite” is a great way to package and distribute a set of add-ons together, for example if you want to distribute Roo custom distributions.

Roo Addon Suite is based on OSGi R5 Subsystems that provides a really convenient deployment model, without compromising the modularity of Roo.

To generate a simple Roo Addon Suite example, you could execute the following command:

```
roo> addon create suite --topLevelPackage org.suite.test --projectName MySuiteProject
--description "New Roo Add-On Suite"
```

This will generate a multi-module project that contains all necessary components to generate and distribute your own "Roo Add-On Suite".

These generated modules are:

- addon-simple: A simple Spring Roo Add-On.
- addon-advanced: An advanced Spring Roo Add-On.
- osgi-bundles: Module that configure bundles.
- roo-addon-suite: Module that generates a .esa file (OSGi R5 Subsystem)
- repository: Module that generates an OSGi Repository ready to distribute your Roo Addon Suite.

15.1. Add-On Simple

This module is a simple Spring Roo Add-On like the explained before. The only difference is that includes an **obr.xml** file.

This obr.xml file is really important in the new Spring Roo arquitechture. This file contains available commands of the add-on and allows Spring Roo to locate add-on on installed repositories when the developer types an add-on command that is not installed yet.

15.2. Add-On Advanced

This module is an advanced Spring Roo Add-On that contains Metadatas and some annotations. As the Add-On Simple, contains an **obr.xml** file that contains available commands of the add-on.

15.3. OSGi Bundles

This module contains only a pom.xml, and it is the responsible of configure all bundles of Spring Roo Add-On suite correctly.

Every Add-On has this bundle as parent on pom.xml.

15.4. Roo Add-On Suite

This module contains necessary maven plugins to generate an OSGi Subsystem in .esa file with all required add-ons.

15.5. Repository

This module generates an OSGi Repository ready to distribute your Roo Addon Suite in your own server after compile.

15.6. Test your Spring Roo Add-On Suite

This generated Spring Roo Add-On Suite contains "suite-dev" file, that allows you to test your generated Roo Add-On Suite using an Spring Roo 2.0+.

Chapter 16. Spring Roo Marketplace

The Spring Roo Marketplace is the alternative to Roobot, easier to maintain and available for everyone, a place to find and keep track on third party addons and Roo Addon Suites.

16.1. Place your Addon Suite in the Roo Marketplace

After develop your own Roo Addon Suite or your own Add-On, you could publish it in your own server using an OSGi Repository as we saw in the point above.

To include your own OSGi Repository on Spring Roo Marketplace page, just send a pull request with the addition to Spring Roo repo's gh-pages branch (roo_addons_suites.html file).

IV. Appendices

The fifth and final part of the reference guide provides appendices and background information that does not neatly belong within the other parts.

The information is intended to be treated as a reference and not read consecutively.

Appendix A: Command Index

This appendix was automatically built from Roo 2.0.0.M2

Commands are listed in alphabetic order, and are shown in monospaced font with any mandatory options you must specify when using the command. Most commands accept a large number of options, and all of the possible options for each command are presented in this appendix.

Chapter 17. Addon Suite Commands

Addon Suite Commands are contained in org.springframework.roo.addon.suite.AddonSuiteCommands.

17.1. addon suite install name

Install some 'Roo Addon Suite' from installed OBR Repository

```
addon suite install name --symbolicName
```

--symbolicName

Name that identifies the 'Roo Addon Suite'; default: '*NULL*' (mandatory)

17.2. addon suite install url

Install some 'Roo Addon Suite' from URL

```
addon suite install url --url
```

--url

URL of Roo Addon Suite .esa file; default: '*NULL*' (mandatory)

17.3. addon suite list

Lists all installed 'Roo Addon Suite'. If you want to list all available 'Roo Addon Suites' on Repository, use --repository parameter

```
addon suite list
```

--repository

OBR Repository where the 'Roo Addon Suite' are located; default: '*NULL*'

17.4. addon suite start

Start some installed 'Roo Addon Suite'

```
addon suite start --symbolicName
```

--symbolicName

Name that identifies the 'Roo Addon Suite'; default: '*NULL*' (mandatory)

17.5. addon suite stop

Stop some started 'Roo Addon Suite'

```
addon suite stop --symbolicName
```

--symbolicName

Name that identifies the 'Roo Addon Suite'; default: '*NULL*' (mandatory)

17.6. addon suite uninstall

Uninstall some installed 'Roo Addon Suite'

```
addon suite uninstall --symbolicName
```

--symbolicName

Name that identifies the 'Roo Addon Suite'; default: '*NULL*' (mandatory)

Chapter 18. Backup Commands

Backup Commands are contained in org.springframework.roo.addon.backup.BackupCommands.

18.1. backup

Backup your project to a zip file

```
backup
```

This command does not accept any options.

Chapter 19. Classpath Commands

Classpath Commands are contained in org.springframework.roo.classpath.operations.ClasspathCommands.

19.1. class

Creates a new Java class source file in any project path

```
class --class
```

--class

The name of the class to create; default: '*NULL*' (mandatory)

--rooAnnotations

Whether the generated class should have common Roo annotations; default if option present: 'true'; default if option not present: 'false'

--path

Source directory to create the class in; default: 'FOCUSED|SRC_MAIN_JAVA'

--extends

The superclass (defaults to java.lang.Object); default if option not present: 'java.lang.Object'

--implements

The interface to implement; default: '*NULL*'

--abstract

Whether the generated class should be marked as abstract; default if option present: 'true'; default if option not present: 'false'

--permitReservedWords

Indicates whether reserved words are ignored by Roo; default if option present: 'true'; default if option not present: 'false'

19.2. constructor

Creates a class constructor

```
constructor
```

--class

The name of the class to receive this constructor; default if option not present: '*'

--fields

The fields to include in the constructor. Multiple field names must be a double-quoted list separated by spaces

19.3. enum constant

Inserts a new enum constant into an enum

```
enum constant --name
```

--class

The name of the enum class to receive this field; default if option not present: '*'

--name

The name of the constant; default: '*NULL*' (mandatory)

--permitReservedWords

Indicates whether reserved words are ignored by Roo; default if option present: 'true'; default if option not present: 'false'

19.4. enum type

Creates a new Java enum source file in any project path

```
enum type --class
```

--class

The name of the enum to create; default: '*NULL*' (mandatory)

--path

Source directory to create the enum in; default: 'FOCUSED|SRC_MAIN_JAVA'

--permitReservedWords

Indicates whether reserved words are ignored by Roo; default if option present: 'true'; default if option not present: 'false'

19.5. focus

Changes focus to a different type

```
focus --class
```

--class

The type to focus on; default: '*NULL*' (mandatory)

19.6. interface

Creates a new Java interface source file in any project path

```
interface --class
```

--class

The name of the interface to create; default: '*NULL*' (mandatory)

--path

Source directory to create the interface in; default: 'FOCUSED|SRC_MAIN_JAVA'

--permitReservedWords

Indicates whether reserved words are ignored by Roo; default if option present: 'true'; default if option not present: 'false'

Chapter 20. Cloud Commands

Cloud Commands are contained in org.springframework.roo.addon.cloud.CloudCommands.

20.1. cloud setup

Setup Cloud Provider on Spring Roo Project

```
cloud setup --provider
```

--provider

Cloud Provider's Name; default: '*NULL*' (mandatory)

--configuration

Plugin Configuration. Add configuration by command like 'key=value,key2=value2,key3=value3'; default: '*NULL*'

Chapter 21. Controller Commands

Controller Commands are contained in org.springframework.roo.addon.web.mvc.controller.addon.ControllerCommands.

21.1. controller all

Scaffold controllers for all project entities without an existing controller - deprecated, use 'web mvc setup' + 'web mvc all' instead

```
controller all --package
```

--package

The package in which new controllers will be placed; default: '*NULL*' (mandatory)

21.2. controller scaffold

Create a new scaffold Controller (ie where we maintain CRUD automatically) - deprecated, use 'web mvc scaffold' instead

```
controller scaffold --class
```

--class

The path and name of the controller object to be created; default: '*NULL*' (mandatory)

--entity

The name of the entity object which the controller exposes to the web tier; default if option not present: '*'

--path

The base path under which the controller listens for RESTful requests (defaults to the simple name of the form backing object); default: '*NULL*'

--disallowedOperations

A comma separated list of operations (only create, update, delete allowed) that should not be generated in the controller; default: '*NULL*'

21.3. web mvc all

Scaffold Spring MVC controllers for all project entities without an existing controller

```
web mvc all --package
```

--package

The package in which new controllers will be placed; default: '*NULL*' (mandatory)

21.4. web mvc scaffold

Create a new scaffold Controller (ie where Roo maintains CRUD functionality automatically)

```
web mvc scaffold --class
```

--class

The path and name of the controller object to be created; default: '*NULL*' (mandatory)

--backingType

The name of the form backing type which the controller exposes to the web tier; default if option not present: '*'

--path

The base path under which the controller listens for RESTful requests (defaults to the simple name of the form backing object); default: '*NULL*'

--disallowedOperations

A comma separated list of operations (only create, update, delete allowed) that should not be generated in the controller; default: '*NULL*'

Chapter 22. Creator Commands

Creator Commands are contained in org.springframework.roo.addon.creator.CreatorCommands.

22.1. addon create advanced

Create a new advanced add-on for Spring Roo (commands + operations
metadata + trigger annotation + dependencies)

```
addon create advanced --topLevelPackage
```

--topLevelPackage

The top level package of the new addon; default: '*NULL*' (mandatory)

--description

Description of your addon (surround text with double quotes); default: '*NULL*'

--projectName

Provide a custom project name (if not provided the top level package name will be used instead); default: '*NULL*'

22.2. addon create i18n

Create a new Internationalization add-on for Spring Roo

```
addon create i18n --topLevelPackage --locale --messageBundle
```

--topLevelPackage

The top level package of the new addon; default: '*NULL*' (mandatory)

--locale

The locale abbreviation (ie: en, or more specific like en_AU, or de_DE); default: '*NULL*' (mandatory)

--messageBundle

Fully qualified path to the messages_xx.properties file; default: '*NULL*' (mandatory)

--language

The full name of the language (used as a label for the UI); default: '*NULL*'

--flagGraphic

Fully qualified path to flag xx.png file; default: '*NULL*'

--description

Description of your addon (surround text with double quotes); default: '*NULL*'

--projectName

Provide a custom project name (if not provided the top level package name will be used instead); default: '*NULL*'

22.3. addon create simple

Create a new simple add-on for Spring Roo (commands + operations)

```
addon create simple --topLevelPackage
```

--topLevelPackage

The top level package of the new addon; default: '*NULL*' (mandatory)

--description

Description of your addon (surround text with double quotes); default: '*NULL*'

--projectName

Provide a custom project name (if not provided the top level package name will be used instead); default: '*NULL*'

22.4. addon create suite

Create a new Spring Roo Addon Suite for Spring Roo (two sample addons
repository + suite generator)

```
addon create suite --topLevelPackage
```

--topLevelPackage

The top level package of all Spring Roo Addon Suite; default: '*NULL*' (mandatory)

--description

Description of your Roo Addon Suite (surround text with double quotes); default: '*NULL*'

--projectName

Provide a custom project name (if not provided the top level package name will be used instead); default: '*NULL*'

22.5. addon create wrapper

Create a new add-on for Spring Roo which wraps a maven artifact to create a OSGi compliant bundle

```
addon create wrapper --topLevelPackage --groupId --artifactId --version --vendorName
--licenseUrl
```

--topLevelPackage

The top level package of the new wrapper bundle; default: '*NULL*' (mandatory)

--groupId

Dependency group id; default: '*NULL*' (mandatory)

--artifactId

Dependency artifact id); default: '*NULL*' (mandatory)

--version

Dependency version; default: '*NULL*' (mandatory)

--vendorName

Dependency vendor name); default: '*NULL*' (mandatory)

--licenseUrl

Dependency license URL; default: '*NULL*' (mandatory)

--docUrl

Dependency documentation URL; default: '*NULL*'

--description

Description of the bundle (use keywords with #-tags for better search integration); default: '*NULL*'

--projectName

Provide a custom project name (if not provided the top level package name will be used instead); default: '*NULL*'

--osgiImports

Contents of Import-Package in OSGi manifest; default: '*NULL*'

Chapter 23. Data On Demand Commands

Data On Demand Commands are contained in org.springframework.roo.addon.dod.addon.DataOnDemandCommands.

23.1. dod

Creates a new data on demand for the specified entity

Chapter 24. dod

--entity

The entity which this data on demand class will create and modify as required; default if option not present: '*'

--class

The class which will be created to hold this data on demand provider (defaults to the entity name + 'DataOnDemand'); default: '*NULL*'

--permitReservedWords

Indicates whether reserved words are ignored by Roo; default if option present: 'true'; default if option not present: 'false'

Chapter 25. Dbre Commands

Dbre Commands are contained in org.springframework.roo.addon.dbre.addon.DbreCommands.

25.1. database introspect

Displays database metadata

```
database introspect --schema
```

--schema

The database schema names. Multiple schema names must be a double-quoted list separated by spaces; default: '*NULL*' (mandatory)

--file

The file to save the metadata to; default: '*NULL*'

--enableViews

Display database views; default if option present: 'true'; default if option not present: 'false'

25.2. database reverse engineer

Create and update entities based on database metadata

```
database reverse engineer --schema
```

--schema

The database schema names. Multiple schema names must be a double-quoted list separated by spaces; default: '*NULL*' (mandatory)

--package

The package in which new entities will be placed; default: '*NULL*'

--testAutomatically

Create automatic integration tests for entities; default if option present: 'true'; default if option not present: 'false'

--enableViews

Reverse engineer database views; default if option present: 'true'; default if option not present: 'false'

--includeTables

The tables to include in reverse engineering. Multiple table names must be a double-quoted list separated by spaces

--excludeTables

The tables to exclude from reverse engineering. Multiple table names must be a double-quoted list separated by spaces

--includeNonPortableAttributes

Include non-portable JPA @Column attributes such as 'columnDefinition'; default if option present: 'true'; default if option not present: 'false'

--disableVersionFields

Disable 'version' field; default if option present: 'true'; default if option not present: 'false'

--disableGeneratedIdentifiers

Disable identifier auto generation; default if option present: 'true'; default if option not present: 'false'

--activeRecord

Generate CRUD active record methods for each entity; default: 'true'

--repository

Generate a repository for each entity; default if option present: 'true'; default if option not present: 'false'

--service

Generate a service for each entity; default if option present: 'true'; default if option not present: 'false'

Chapter 26. Embedded Commands

Embedded Commands are contained in org.springframework.roo.addon.web.mvc.embedded.EmbeddedCommands.

26.1. web mvc embed document

Embed a document for your WEB MVC application

```
web mvc embed document --provider --documentId
```

--provider

The id of the document; default: '*NULL*' (mandatory)

--documentId

The id of the document; default: '*NULL*' (mandatory)

--viewName

The name of the jspx view; default: '*NULL*'

26.2. web mvc embed generic

Embed media by URL into your WEB MVC application

```
web mvc embed generic --url
```

--url

The url of the source to be embedded; default: '*NULL*' (mandatory)

--viewName

The name of the jspx view; default: '*NULL*'

26.3. web mvc embed map

Embed a map for your WEB MVC application

```
web mvc embed map --location
```

--location

The location of the map (ie "Sydney, Australia"); default: '*NULL*' (mandatory)

--viewName

The name of the jspx view; default: '*NULL*'

26.4. web mvc embed photos

Embed a photo gallery for your WEB MVC application

```
web mvc embed photos --provider --userId --albumId
```

--provider

The provider of the photo gallery; default: '*NULL*' (mandatory)

--userId

The user id; default: '*NULL*' (mandatory)

--albumId

The album id; default: '*NULL*' (mandatory)

--viewName

The name of the jspx view; default: '*NULL*'

26.5. web mvc embed stream video

Embed a video stream into your WEB MVC application

```
web mvc embed stream video --provider --streamId
```

--provider

The provider of the video stream; default: '*NULL*' (mandatory)

--streamId

The stream id; default: '*NULL*' (mandatory)

--viewName

The name of the jspx view; default: '*NULL*'

26.6. web mvc embed twitter

Embed twitter messages into your WEB MVC application

```
web mvc embed twitter --searchTerm
```

--searchTerm

The search term to display results for; default: '*NULL*' (mandatory)

--viewName

The name of the jspx view; default: '*NULL*'

26.7. web mvc embed video

Embed a video for your WEB MVC application

```
web mvc embed video --provider --videoId
```

--provider

The id of the video; default: '*NULL*' (mandatory)

--videoId

The id of the video; default: '*NULL*' (mandatory)

--viewName

The name of the jspx view; default: '*NULL*'

26.8. web mvc embed wave

Embed Google wave integration for your WEB MVC application

```
web mvc embed wave --waveId
```

--waveId

The key of the wave; default: '*NULL*' (mandatory)

--viewName

The name of the jspx view; default: '*NULL*'

Chapter 27. Equals Commands

Equals Commands are contained in org.springframework.roo.addon.javabean.addon.EqualsCommands.

27.1. equals

Add equals and hashCode methods to a class

```
equals
```

--class

The name of the class; default if option not present: '*'

--appendSuper

Whether to call the super class equals and hashCode methods; default if option present: 'true'; default if option not present: 'false'

--excludeFields

The fields to exclude in the equals and hashcode methods. Multiple field names must be a double-quoted list separated by spaces

Chapter 28. Felix Delegator

Felix Delegator are contained in org.springframework.roo.felix.FelixDelegator.

28.1. !g

Passes a command directly through to the Felix shell infrastructure

!g

--[default]

The command to pass to Felix (WARNING: no validation or security checks are performed); default: 'help'

28.2. exit

Exits the shell

```
exit
```

This command does not accept any options.

Chapter 29. Field Commands

Field Commands are contained in org.springframework.roo.classpath.operations.FieldCommands.

29.1. field boolean

Adds a private boolean field to an existing Java source file

```
field boolean --fieldName
```

--fieldName

The name of the field to add; default: '*NULL*' (mandatory)

--class

The name of the class to receive this field; default if option not present: '*'

--notNull

Whether this value cannot be null; default if option present: 'true'; default if option not present: 'false'

--nullRequired

Whether this value must be null; default if option present: 'true'; default if option not present: 'false'

--assertFalse

Whether this value must assert false; default if option present: 'true'; default if option not present: 'false'

--assertTrue

Whether this value must assert true; default if option present: 'true'; default if option not present: 'false'

--column

The JPA @Column name; default: '*NULL*'

--value

Inserts an optional Spring @Value annotation with the given content; default: '*NULL*'

--comment

An optional comment for JavaDocs; default: '*NULL*'

--primitive

Indicates to use a primitive type; default if option present: 'true'; default if option not present: 'false'

--transient

Indicates to mark the field as transient; default if option present: 'true'; default if option not present: 'false'

--permitReservedWords

Indicates whether reserved words are ignored by Roo; default if option present: 'true'; default if option not present: 'false'

29.2. field date

Adds a private date field to an existing Java source file

```
field date --fieldName --type
```

--fieldName

The name of the field to add; default: '*NULL*' (mandatory)

--type

The Java type of the entity; default: '*NULL*' (mandatory)

--persistenceType

The type of persistent storage to be used; default: '*NULL*'

--class

The name of the class to receive this field; default if option not present: '*'

--notNull

Whether this value cannot be null; default if option present: 'true'; default if option not present: 'false'

--nullRequired

Whether this value must be null; default if option present: 'true'; default if option not present: 'false'

--future

Whether this value must be in the future; default if option present: 'true'; default if option not present: 'false'

--past

Whether this value must be in the past; default if option present: 'true'; default if option not present: 'false'

--column

The JPA @Column name; default: '*NULL*'

--comment

An optional comment for JavaDocs; default: '*NULL*'

--value

Inserts an optional Spring @Value annotation with the given content; default: '*NULL*'

--transient

Indicates to mark the field as transient; default if option present: 'true'; default if option not present: 'false'

--permitReservedWords

Indicates whether reserved words are ignored by Roo; default if option present: 'true'; default if option not present: 'false'

--dateFormat

Indicates the style of the date format (ignored if dateTimeFormatPattern is specified); default: 'MEDIUM'

--timeFormat

Indicates the style of the time format (ignored if dateTimeFormatPattern is specified); default: 'NONE'

--dateTimeFormatPattern

Indicates a DateTime format pattern such as yyyy-MM-dd hh:mm:ss a; default: '*NULL*'

29.3. field embedded

Adds a private @Embedded field to an existing Java source file

```
field embedded --fieldName --type
```

--fieldName

The name of the field to add; default: '*NULL*' (mandatory)

--type

The Java type of the @Embeddable class; default: '*NULL*' (mandatory)

--class

The name of the @Entity class to receive this field; default if option not present: '*'

--permitReservedWords

Indicates whether reserved words are ignored by Roo; default if option present: 'true'; default if option not present: 'false'

29.4. field enum

Adds a private enum field to an existing Java source file

```
field enum --fieldName --type
```

--fieldName

The name of the field to add; default: '*NULL*' (mandatory)

--type

The enum type of this field; default: '*NULL*' (mandatory)

--class

The name of the class to receive this field; default if option not present: '*'

--column

The JPA @Column name; default: '*NULL*'

--notNull

Whether this value cannot be null; default if option present: 'true'; default if option not present: 'false'

--nullRequired

Whether this value must be null; default if option present: 'true'; default if option not present: 'false'

--enumType

The fetch semantics at a JPA level; default: '*NULL*'

--comment

An optional comment for JavaDocs; default: '*NULL*'

--transient

Indicates to mark the field as transient; default if option present: 'true'; default if option not present: 'false'

--permitReservedWords

Indicates whether reserved words are ignored by Roo; default if option present: 'true'; default if option not present: 'false'

29.5. field file

Adds a byte array field for storing uploaded file contents (JSF-scaffolded UIs only)

```
field file --fieldName --contentType
```

--fieldName

The name of the file upload field to add; default: '*NULL*' (mandatory)

--class

The name of the class to receive this field; default if option not present: '*'

--contentType

The content type of the file; default: '*NULL*' (mandatory)

--autoUpload

Whether the file is uploaded automatically when selected; default if option present: 'true'; default if option not present: 'false'

--column

The JPA @Column name; default: '*NULL*'

--notNull

Whether this value cannot be null; default if option present: 'true'; default if option not present: 'false'

--permitReservedWords

Indicates whether reserved words are ignored by Roo; default if option present: 'true'; default if option not present: 'false'

29.6. field list

Adds a private List field to an existing Java source file (eg the 'one' side of a many-to-one)

```
field list --fieldName --type
```

--fieldName

The name of the field to add; default: '*NULL*' (mandatory)

--type

The entity which will be contained within the Set; default: '*NULL*' (mandatory)

--class

The name of the class to receive this field; default if option not present: '*'

--mappedBy

The field name on the referenced type which owns the relationship; default: '*NULL*'

--notNull

Whether this value cannot be null; default if option present: 'true'; default if option not present: 'false'

--nullRequired

Whether this value must be null; default if option present: 'true'; default if option not present: 'false'

--sizeMin

The minimum number of elements in the collection; default: '*NULL*'

--sizeMax

The maximum number of elements in the collection; default: '*NULL*'

--cardinality

The relationship cardinality at a JPA level; default: 'MANY_TO_MANY'

--fetch

The fetch semantics at a JPA level; default: '*NULL*'

--comment

An optional comment for JavaDocs; default: '*NULL*'

--transient

Indicates to mark the field as transient; default if option present: 'true'; default if option not present: 'false'

--permitReservedWords

Indicates whether reserved words are ignored by Roo; default if option present: 'true'; default if option not present: 'false'

29.7. field number

Adds a private numeric field to an existing Java source file

```
field number --fieldName --type
```

--fieldName

The name of the field to add; default: '*NULL*' (mandatory)

--type

The Java type of the entity; default: '*NULL*' (mandatory)

--class

The name of the class to receive this field; default if option not present: '*'

--notNull

Whether this value cannot be null; default if option present: 'true'; default if option not present: 'false'

--nullRequired

Whether this value must be null; default if option present: 'true'; default if option not present: 'false'

--decimalMin

The BigDecimal string-based representation of the minimum value; default: '*NULL*'

--decimalMax

The BigDecimal string based representation of the maximum value; default: '*NULL*'

--digitsInteger

Maximum number of integral digits accepted for this number; default: '*NULL*'

--digitsFraction

Maximum number of fractional digits accepted for this number; default: '*NULL*'

--min

The minimum value; default: '*NULL*'

--max

The maximum value; default: '*NULL*'

--column

The JPA @Column name; default: '*NULL*'

--comment

An optional comment for JavaDocs; default: '*NULL*'

--value

Inserts an optional Spring @Value annotation with the given content; default: '*NULL*'

--transient

Indicates to mark the field as transient; default if option present: 'true'; default if option not present: 'false'

--primitive

Indicates to use a primitive type if possible; default if option present: 'true'; default if option not present: 'false'

--unique

Indicates whether to mark the field with a unique constraint; default if option present: 'true'; default if option not present: 'false'

--permitReservedWords

Indicates whether reserved words are ignored by Roo; default if option present: 'true'; default if option not present: 'false'

29.8. field other

Inserts a private field into the specified file

```
field other --fieldName --type
```

--fieldName

The name of the field; default: '*NULL*' (mandatory)

--type

The Java type of this field; default: '*NULL*' (mandatory)

--class

The name of the class to receive this field; default if option not present: '*'

--notNull

Whether this value cannot be null; default if option present: 'true'; default if option not present: 'false'

--nullRequired

Whether this value must be null; default if option present: 'true'; default if option not present: 'false'

--comment

An optional comment for JavaDocs; default: '*NULL*'

--column

The JPA @Column name; default: '*NULL*'

--value

Inserts an optional Spring @Value annotation with the given content; default: '*NULL*'

--transient

Indicates to mark the field as transient; default if option present: 'true'; default if option not present: 'false'

--permitReservedWords

Indicates whether reserved words are ignored by Roo; default if option present: 'true'; default if option not present: 'false'

29.9. field reference

Adds a private reference field to an existing Java source file (eg the 'many' side of a many-to-one)

```
field reference --fieldName --type
```

--fieldName

The name of the field to add; default: '*NULL*' (mandatory)

--type

The Java type of the entity to reference; default: '*NULL*' (mandatory)

--class

The name of the class to receive this field; default if option not present: '*'

--notNull

Whether this value cannot be null; default if option present: 'true'; default if option not present: 'false'

--nullRequired

Whether this value must be null; default if option present: 'true'; default if option not present: 'false'

--joinColumnName

The JPA @JoinColumn name; default: '*NULL*'

--referencedColumnName

The JPA @JoinColumn referencedColumnName; default: '*NULL*'

--cardinality

The relationship cardinality at a JPA level; default: 'MANY_TO_ONE'

--fetch

The fetch semantics at a JPA level; default: '*NULL*'

--comment

An optional comment for JavaDocs; default: '*NULL*'

--transient

Indicates to mark the field as transient; default if option present: 'true'; default if option not present: 'false'

--permitReservedWords

Indicates whether reserved words are ignored by Roo; default if option present: 'true'; default if option not present: 'false'

29.10. field set

Adds a private Set field to an existing Java source file (eg the 'one' side of a many-to-one)

```
field set --fieldName --type
```

--fieldName

The name of the field to add; default: '*NULL*' (mandatory)

--type

The entity which will be contained within the Set; default: '*NULL*' (mandatory)

--class

The name of the class to receive this field; default if option not present: '*'

--mappedBy

The field name on the referenced type which owns the relationship; default: '*NULL*'

--notNull

Whether this value cannot be null; default if option present: 'true'; default if option not present: 'false'

--nullRequired

Whether this value must be null; default if option present: 'true'; default if option not present: 'false'

--sizeMin

The minimum number of elements in the collection; default: '*NULL*'

--sizeMax

The maximum number of elements in the collection; default: '*NULL*'

--cardinality

The relationship cardinality at a JPA level; default: 'MANY_TO_MANY'

--fetch

The fetch semantics at a JPA level; default: '*NULL*'

--comment

An optional comment for JavaDocs; default: '*NULL*'

--transient

Indicates to mark the field as transient; default if option present: 'true'; default if option not present: 'false'

--permitReservedWords

Indicates whether reserved words are ignored by Roo; default if option present: 'true'; default if option not present: 'false'

29.11. field string

Adds a private string field to an existing Java source file

```
field string --fieldName
```

--fieldName

The name of the field to add; default: '*NULL*' (mandatory)

--class

The name of the class to receive this field; default if option not present: '*'

--notNull

Whether this value cannot be null; default if option present: 'true'; default if option not present: 'false'

--nullRequired

Whether this value must be null; default if option present: 'true'; default if option not present: 'false'

--decimalMin

The BigDecimal string-based representation of the minimum value; default: '*NULL*'

--decimalMax

The BigDecimal string based representation of the maximum value; default: '*NULL*'

--sizeMin

The minimum string length; default: '*NULL*'

--sizeMax

The maximum string length; default: '*NULL*'

--regexp

The required regular expression pattern; default: '*NULL*'

--column

The JPA @Column name; default: '*NULL*'

--value

Inserts an optional Spring @Value annotation with the given content; default: '*NULL*'

--comment

An optional comment for JavaDocs; default: '*NULL*'

--transient

Indicates to mark the field as transient; default if option present: 'true'; default if option not present: 'false'

--unique

Indicates whether to mark the field with a unique constraint; default if option present: 'true'; default if option not present: 'false'

--permitReservedWords

Indicates whether reserved words are ignored by Roo; default if option present: 'true'; default if option not present: 'false'

--lob

Indicates that this field is a Large Object; default if option present: 'true'; default if option not present: 'false'

Chapter 30. Finder Commands

Finder Commands are contained in org.springframework.roo.addon.finder.FinderCommands.

30.1. finder add

Install finders in the given target (must be an entity)

```
finder add --finderName
```

--class

The controller or entity for which the finders are generated; default if option not present: '*'

--finderName

The finder string as generated with the 'finder list' command; default: '*NULL*' (mandatory)

30.2. finder list

List all finders for a given target (must be an entity)

```
finder list
```

--class

The controller or entity for which the finders are generated; default if option not present: '*'

--depth

The depth of attribute combinations to be generated for the finders; default: '1'

--filter

A comma separated list of strings that must be present in a filter to be included; default: '*NULL*'

Chapter 31. Help Commands

Help Commands are contained in org.springframework.roo.felix.help.HelpCommands.

31.1. help

Shows system help

```
help
```

--command

Command name to provide help for; default: '*NULL*'

31.2. reference guide

Writes the reference guide XML fragments (in DocBook format) into the current working directory

```
reference guide
```

This command does not accept any options.

Chapter 32. Hint Commands

Hint Commands are contained in org.springframework.roo.classpath.operations.HintCommands.

32.1. hint

Provides step-by-step hints and context-sensitive guidance

```
hint
```

--topic

The topic for which advice should be provided

Chapter 33. Integration Test Commands

Integration Test Commands are contained in org.springframework.roo.addon.test.addon.IntegrationTestCommands.

33.1. test integration

Creates a new integration test for the specified entity

```
test integration
```

--entity

The name of the entity to create an integration test for; default if option not present: '*'

--permitReservedWords

Indicates whether reserved words are ignored by Roo; default if option present: 'true'; default if option not present: 'false'

--transactional

Indicates whether the created test cases should be run withing a Spring transaction; default: 'true'

33.2. test mock

Creates a mock test for the specified entity

```
test mock
```

--entity

The name of the entity this mock test is targeting; default if option not present: '*'

--permitReservedWords

Indicates whether reserved words are ignored by Roo; default if option present: 'true'; default if option not present: 'false'

33.3. test stub

Creates a test stub for the specified class

```
test stub
```

--class

The name of the class this mock test is targeting; default if option not present: '*'

--permitReservedWords

Indicates whether reserved words are ignored by Roo; default if option present: 'true'; default if option not present: 'false'

Chapter 34. J Line Shell Component

J Line Shell Component are contained in org.springframework.roo.shell.jline.osgi.JLineShellComponent.

*/ ~~

End of block comment

*/

This command does not accept any options.

/* ~~

Start of block comment

/*

This command does not accept any options.

~~

Inline comment markers (start of line only)

This command does not accept any options.

34.1. date

Displays the local date and time

```
date
```

This command does not accept any options.

34.2. script

Parses the specified resource file and executes its commands

```
script --file
```

--file

The file to locate and execute; default: '*NULL*' (mandatory)

--lineNumbers

Display line numbers when executing the script; default if option present: 'true'; default if option not present: 'false'

34.3. system properties

Shows the shell's properties

```
system properties
```

This command does not accept any options.

34.4. version

Displays shell version

```
version
```

--[default]

Special version flags; default: '*NULL*'

Chapter 35. Jms Commands

Jms Commands are contained in org.springframework.roo.addon.jms.JmsCommands.

35.1. field jms template

Insert a JmsOperations field into an existing type

```
field jms template
```

--fieldName

The name of the field to add; default: 'jmsOperations'

--class

The name of the class to receive this field; default if option not present: '*'

--async

Indicates if the injected method should be executed asynchronously; default if option present: 'true'; default if option not present: 'false'

35.2. jms listener class

Create an asynchronous JMS consumer

```
jms listener class --class
```

--class

The name of the class to create; default: '*NULL*' (mandatory)

--destinationName

The name of the destination; default: 'myDestination'

--destinationType

The type of the destination; default: 'QUEUE'

35.3. jms setup

Install a JMS provider into your project

```
jms setup --provider
```

--provider

The persistence provider to support; default: '*NULL*' (mandatory)

--destinationName

The name of the destination; default: 'myDestination'

--destinationType

The type of the destination; default: 'QUEUE'

Chapter 36. Jpa Commands

Jpa Commands are contained in org.springframework.roo.addon.jpa.addon.JpaCommands.

36.1. database properties list

Shows database configuration details

```
database properties list
```

This command does not accept any options.

36.2. database properties remove

Removes a particular database property

```
database properties remove --key
```

--key

The property key that should be removed; default: '*NULL*' (mandatory)

36.3. database properties set

Changes a particular database property

```
database properties set --key --value
```

--key

The property key that should be changed; default: '*NULL*' (mandatory)

--value

The new vale for this property key; default: '*NULL*' (mandatory)

36.4. embeddable

Creates a new Java class source file with the JPA @Embeddable annotation in SRC_MAIN_JAVA

```
embeddable --class
```

--class

The name of the class to create; default: '*NULL*' (mandatory)

--serializable

Whether the generated class should implement java.io.Serializable; default if option present: 'true'; default if option not present: 'false'

--permitReservedWords

Indicates whether reserved words are ignored by Roo; default if option present: 'true'; default if option not present: 'false'

36.5. entity jpa

Creates a new JPA persistent entity in SRC_MAIN_JAVA

```
entity jpa --class
```

--class

Name of the entity to create; default: '*NULL*' (mandatory)

--extends

The superclass (defaults to java.lang.Object); default if option not present: 'java.lang.Object'

--implements

The interface to implement; default: '*NULL*'

--abstract

Whether the generated class should be marked as abstract; default if option present: 'true'; default if option not present: 'false'

--testAutomatically

Create automatic integration tests for this entity; default if option present: 'true'; default if option not present: 'false'

--table

The JPA table name to use for this entity; default: '*NULL*'

--schema

The JPA table schema name to use for this entity; default: '*NULL*'

--catalog

The JPA table catalog name to use for this entity; default: '*NULL*'

--identifierField

The JPA identifier field name to use for this entity; default: '*NULL*'

--identifierColumn

The JPA identifier field column to use for this entity; default: '*NULL*'

--identifierType

The data type that will be used for the JPA identifier field (defaults to java.lang.Long); default: 'java.lang.Long'

--versionField

The JPA version field name to use for this entity; default: '*NULL*'

--versionColumn

The JPA version field column to use for this entity; default: '*NULL*'

--versionType

The data type that will be used for the JPA version field (defaults to java.lang.Integer); default if option not present: 'java.lang.Integer'

--inheritanceType

The JPA @Inheritance value (apply to base class); default: '*NULL*'

--mappedSuperclass

Apply @MappedSuperclass for this entity; default if option present: 'true'; default if option not present: 'false'

--equals

Whether the generated class should implement equals and hashCode methods; default if option present: 'true'; default if option not present: 'false'

--serializable

Whether the generated class should implement java.io.Serializable; default if option present: 'true'; default if option not present: 'false'

--persistenceUnit

The persistence unit name to be used in the persistence.xml file; default: '*NULL*'

--transactionManager

The transaction manager name; default: '*NULL*'

--permitReservedWords

Indicates whether reserved words are ignored by Roo; default if option present: 'true'; default if option not present: 'false'

--entityName

The name used to refer to the entity in queries; default: '*NULL*'

--sequenceName

The name of the sequence for incrementing sequence-driven primary keys; default: '*NULL*'

--activeRecord

Generate CRUD active record methods for this entity; default: 'true'

36.6. jpa setup

Install or updates a JPA persistence provider in your project

```
jpa setup --provider --database
```

--provider

The persistence provider to support; default: '*NULL*' (mandatory)

--database

The database to support; default: '*NULL*' (mandatory)

--applicationId

The Google App Engine application identifier to use; default if option not present: 'the project's name'

--jndiDataSource

The JNDI datasource to use; default: '*NULL*'

--hostName

The host name to use; default: '*NULL*'

--databaseName

The database name to use; default: '*NULL*'

--userName

The username to use; default: '*NULL*'

--password

The password to use; default: '*NULL*'

--transactionManager

The transaction manager name; default: '*NULL*'

--persistenceUnit

The persistence unit name to be used in the persistence.xml file; default: '*NULL*'

36.7. persistence setup

Install or updates a JPA persistence provider in your project - deprecated, use 'jpa setup' instead

```
persistence setup --provider --database
```

--provider

The persistence provider to support; default: '*NULL*' (mandatory)

--database

The database to support; default: '*NULL*' (mandatory)

--applicationId

The Google App Engine application identifier to use; default if option not present: 'the project's name'

--jndiDataSource

The JNDI datasource to use; default: '*NULL*'

--hostName

The host name to use; default: '*NULL*'

--databaseName

The database name to use; default: '*NULL*'

--userName

The username to use; default: '*NULL*'

--password

The password to use; default: '*NULL*'

--transactionManager

The transaction manager name; default: '*NULL*'

--persistenceUnit

The persistence unit name to be used in the persistence.xml file; default: '*NULL*'

Chapter 37. Json Commands

Json Commands are contained in org.springframework.roo.addon.json.addon.JsonCommands.

37.1. json add

Adds @RooJson annotation to target type

```
json add
```

--class

The java type to apply this annotation to; default if option not present: '*'

--rootName

The root name which should be used to wrap the JSON document; default: '*NULL*'

--deepSerialize

Indication if deep serialization should be enabled.; default if option present: 'true'; default if option not present: 'false'

--iso8601Dates

Indication if dates should be formatted according to ISO 8601; default if option present: 'true'; default if option not present: 'false'

37.2. json all

Adds @RooJson annotation to all types annotated with @RooJavaBean

```
json all
```

--deepSerialize

Indication if deep serialization should be enabled; default if option present: 'true'; default if option not present: 'false'

--iso8601Dates

Indication if dates should be formatted according to ISO 8601; default if option present: 'true'; default if option not present: 'false'

Chapter 38. Jsp Commands

Jsp Commands are contained in org.springframework.roo.addon.web.mvc.jsp.JspCommands.

38.1. controller class

Create a new manual Controller (ie where you write the methods) - deprecated, use 'web mvc controller' instead

```
controller class --class
```

--class

The path and name of the controller object to be created; default: '*NULL*' (mandatory)

--preferredMapping

Indicates a specific request mapping path for this controller (eg /foo/); default: '*NULL*'

38.2. web mvc controller

Create a new manual Controller (ie where you write the methods)

```
web mvc controller --class
```

--class

The path and name of the controller object to be created; default: '*NULL*' (mandatory)

--preferredMapping

Indicates a specific request mapping path for this controller (eg /foo/); default: '*NULL*'

38.3. web mvc install language

Install new internationalization bundle for MVC scaffolded UI.

```
web mvc install language --code
```

--code

The language code for the desired bundle; default: '*NULL*' (mandatory)

38.4. web mvc install view

Create a new static view.

```
web mvc install view --path --viewName --title
```

--path

The path the static view to create in (required, ie '/foo/blah'); default: '*NULL*' (mandatory)

--viewName

The view name the mapping this view should adopt (required, ie 'index'); default: '*NULL*' (mandatory)

--title

The title of the view; default: '*NULL*' (mandatory)

38.5. web mvc language

Install new internationalization bundle for MVC scaffolded UI.

```
web mvc language --code
```

--code

The language code for the desired bundle; default: '*NULL*' (mandatory)

38.6. web mvc setup

Setup a basic project structure for a Spring MVC / JSP application

```
web mvc setup
```

This command does not accept any options.

38.7. web mvc update tags

Replace an existing application tagx library with the latest version (use --backup option to backup your application first)

```
web mvc update tags
```

--backup

Backup your application before replacing your existing tag library; default if option present: 'true'; default if option not present: 'false'

38.8. web mvc view

Create a new static view.

```
web mvc view --path --viewName --title
```

--path

The path the static view to create in (required, ie '/foo/blah'); default: '*NULL*' (mandatory)

--viewName

The view name the mapping this view should adopt (required, ie 'index'); default: '*NULL*' (mandatory)

--title

The title of the view; default: '*NULL*' (mandatory)

Chapter 39. Logging Commands

Logging Commands are contained in org.springframework.roo.addon.logging.LoggingCommands.

39.1. logging setup

Configure logging in your project

```
logging setup --level
```

--level

The log level to configure; default: '*NULL*' (mandatory)

--package

The package to append the logging level to (all by default); default: '*NULL*'

Chapter 40. Mail Commands

Mail Commands are contained in org.springframework.roo.addon.email.MailCommands.

40.1. email sender setup

Install a Spring JavaMailSender in your project

```
email sender setup --hostServer
```

--hostServer

The host server; default: '*NULL*' (mandatory)

--protocol

The protocol used by mail server; default: '*NULL*'

--port

The port used by mail server; default: '*NULL*'

--encoding

The encoding used for mail; default: '*NULL*'

--username

The mail account username; default: '*NULL*'

--password

The mail account password; default: '*NULL*'

40.2. email template setup

Configures a template for a SimpleMailMessage

```
email template setup
```

--from

The 'from' email (optional); default: '*NULL*'

--subject

The message subject (obtional); default: '*NULL*'

40.3. field email template

Inserts a MailTemplate field into an existing type

```
field email template
```

--fieldName

The name of the field to add; default: 'mailTemplate'

--class

The name of the class to receive this field; default if option not present: '*'

--async

Indicates if the injected method should be executed asynchronously; default if option present: 'true'; default if option not present: 'false'

Chapter 41. Maven Commands

Maven Commands are contained in org.springframework.roo.project.MavenCommands.

41.1. dependency add

Adds a new dependency to the Maven project object model (POM)

```
dependency add --groupId --artifactId --version
```

--groupId

The group ID of the dependency; default: '*NULL*' (mandatory)

--artifactId

The artifact ID of the dependency; default: '*NULL*' (mandatory)

--version

The version of the dependency; default: '*NULL*' (mandatory)

--classifier

The classifier of the dependency; default: '*NULL*'

--scope

The scope of the dependency; default: '*NULL*'

41.2. dependency remove

Removes an existing dependency from the Maven project object model (POM)

```
dependency remove --groupId --artifactId --version
```

--groupId

The group ID of the dependency; default: '*NULL*' (mandatory)

--artifactId

The artifact ID of the dependency; default: '*NULL*' (mandatory)

--version

The version of the dependency; default: '*NULL*' (mandatory)

--classifier

The classifier of the dependency; default: '*NULL*'

41.3. maven repository add

Adds a new repository to the Maven project object model (POM)

```
maven repository add --id --url
```

--id

The ID of the repository; default: '*NULL*' (mandatory)

--name

The name of the repository; default: '*NULL*'

--url

The URL of the repository; default: '*NULL*' (mandatory)

41.4. maven repository remove

Removes an existing repository from the Maven project object model (POM)

```
maven repository remove --id --url
```

--id

The ID of the repository; default: '*NULL*' (mandatory)

--url

The URL of the repository; default: '*NULL*' (mandatory)

41.5. module create

Creates a new Maven module

```
module create --moduleName --topLevelPackage
```

--moduleName

The name of the module; default: '*NULL*' (mandatory)

--topLevelPackage

The uppermost package name (this becomes the <groupId> in Maven and also the '~' value when using Roo's shell); default: '*NULL*' (mandatory)

--java

Forces a particular major version of Java to be used (will be auto-detected if unspecified; specify 6 or 7 only); default: '*NULL*'

--parent

The Maven coordinates of the parent POM, in the form "groupId:artifactId:version"; default: '*NULL*'

--packaging

The Maven packaging of this module; default if option not present: 'jar'

--artifactId

The artifact ID of this module (defaults to moduleName if not specified); default: '*NULL*'

41.6. module focus

Changes focus to a different project module

```
module focus --moduleName
```

--moduleName

The module to focus on; default: '*NULL*' (mandatory)

41.7. perform assembly

Executes the assembly goal via Maven

```
perform assembly
```

This command does not accept any options.

41.8. perform clean

Executes a full clean (including Eclipse files) via Maven

```
perform clean
```

This command does not accept any options.

41.9. perform command

Executes a user-specified Maven command

```
perform command --mavenCommand
```

--mavenCommand

User-specified Maven command (eg test:test); default: '*NULL*' (mandatory)

41.10. perform eclipse

Sets up Eclipse configuration via Maven (only necessary if you have not installed the m2eclipse plugin in Eclipse)

```
perform eclipse
```

This command does not accept any options.

41.11. perform package

Packages the application using Maven, but does not execute any tests

```
perform package
```

This command does not accept any options.

41.12. perform tests

Executes the tests via Maven

```
perform tests
```

This command does not accept any options.

Chapter 42. Metadata Commands

Metadata Commands are contained in org.springframework.roo.classpath.MetadataCommands.

42.1. metadata cache

Shows detailed metadata for the indicated type

```
metadata cache --maximumCapacity
```

--maximumCapacity

The maximum number of metadata items to cache; default: '*NULL*' (mandatory)

42.2. metadata for id

Shows detailed information about the metadata item

```
metadata for id --metadataId
```

--metadataId

The metadata ID (should start with MID:); default: '*NULL*' (mandatory)

42.3. metadata for module

Shows the ProjectMetadata for the indicated project module

```
metadata for module
```

--module

The module for which to retrieve the metadata (defaults to the focused module); default: '*NULL*'

42.4. metadata for type

Shows detailed metadata for the indicated type

```
metadata for type --type
```

--type

The Java type for which to display metadata; default: '*NULL*' (mandatory)

42.5. metadata status

Shows metadata statistics

```
metadata status
```

This command does not accept any options.

42.6. metadata trace

Traces metadata event delivery notifications

```
metadata trace --level
```

--level

The verbosity of notifications (0=none, 1=some, 2=all); default: '*NULL*' (mandatory)

Chapter 43. Obr Add On Commands

Obr Add On Commands are contained in org.springframework.roo.obr.addon.search.ObrAddOnCommands.

43.1. addon info bundle

Provide information about a specific Spring Roo Add-on

```
addon info bundle --bundleSymbolicName
```

--bundleSymbolicName

The bundle symbolic name for the add-on of interest; default: '*NULL*' (mandatory)

43.2. addon install bundle

Install Spring Roo Add-on

```
addon install bundle --bundleSymbolicName
```

--bundleSymbolicName

The bundle symbolic name for the add-on of interest; default: '*NULL*' (mandatory)

43.3. addon install url

Install Spring Roo Add-on using url

```
addon install url --url
```

--url

The url for the add-on of interest; default: '*NULL*' (mandatory)

43.4. addon list

List all installed addons

```
addon list
```

This command does not accept any options.

43.5. addon remove

Remove Spring Roo Add-on

```
addon remove --bundleSymbolicName
```

--bundleSymbolicName

The bundle symbolic name for the add-on of interest; default: '*NULL*' (mandatory)

43.6. addon search

Search all known Spring Roo Add-ons

```
addon search --requiresCommand
```

--requiresCommand

Only display add-ons in search results that offer this command; default: '*NULL*' (mandatory)

Chapter 44. Obr Repository Commands

Obr Repository Commands are contained in org.springframework.roo.obr.addon.search.ObrRepositoryCommands.

44.1. addon repository add

Adds a new OBR Repository to ROO Shell

```
addon repository add --url
```

--url

URL file that defines repository. Ex: 'http://localhost/repo/index.xml'; default: '*NULL*' (mandatory)

44.2. addon repository introspect

Introspects all installed OBR Repositories and list all their addons

```
addon repository introspect
```

This command does not accept any options.

44.3. addon repository list

Lists existing OBR Repositories

```
addon repository list
```

This command does not accept any options.

44.4. addon repository remove

Removes an existing OBR Repository from ROO Shell

```
addon repository remove --url
```

--url

URL file that defines repository. Ex: 'http://localhost/repo/index.xml'; default: '*NULL*' (mandatory)

Chapter 45. Os Commands

Os Commands are contained in org.springframework.roo.addon.oscommands.OsCommands.

! ~

Allows execution of operating system (OS) commands.

- ! -

--command

The command to execute; default: ''

Chapter 46. Pgp Commands

Pgp Commands are contained in org.springframework.roo.felix.pgp.PgpCommands.

46.1. pgp automatic trust

Indicates to automatically trust all keys encountered until the command is invoked again

```
pgp automatic trust
```

This command does not accept any options.

46.2. pgp key view

Downloads a remote key and displays it to the user (does not change any trusts)

```
pgp key view --keyId
```

--keyId

The key ID to view (eg 00B5050F or 0x00B5050F); default: '*NULL*' (mandatory)

46.3. pgp list trusted keys

Lists the keys you currently trust and have not been revoked at the time last downloaded from a public key server

```
pgp list trusted keys
```

This command does not accept any options.

46.4. pgp refresh all

Refreshes all keys from public key servers

```
pgp refresh all
```

This command does not accept any options.

46.5. pgp status

Displays the status of the PGP environment

```
pgp status
```

This command does not accept any options.

46.6. pgp trust

Grants trust to a particular key ID

```
pgp trust --keyId
```

--keyId

The key ID to trust (eg 00B5050F or 0x00B5050F); default: '*NULL*' (mandatory)

46.7. pgp untrust

Revokes your trust for a particular key ID

```
pgp untrust --keyId
```

--keyId

The key ID to remove trust from (eg 00B5050F or 0x00B5050F); default: '*NULL*' (mandatory)

Chapter 47. Process Manager Diagnostics Listener

Process Manager Diagnostics Listener are contained in org.springframework.roo.process.manager.internal.ProcessManagerDiagnosticsListener.

47.1. process manager debug

Indicates if process manager debugging is desired

```
process manager debug
```

--enabled

Activates debug mode; default: 'true'

Chapter 48. Project Commands

Project Commands are contained in org.springframework.roo.project.ProjectCommands.

48.1. development mode

Switches the system into development mode (greater diagnostic information)

```
development mode
```

--enabled

Activates development mode; default: 'true'

48.2. project scan now

Perform a manual file system scan

```
project scan now
```

This command does not accept any options.

48.3. project scan speed

Changes the file system scanning speed

```
project scan speed --ms
```

--ms

The number of milliseconds between each scan; default: '*NULL*' (mandatory)

48.4. project scan status

Display file system scanning information

```
project scan status
```

This command does not accept any options.

48.5. project setup

Creates a new Maven project

```
project setup --topLevelPackage
```

--topLevelPackage

The uppermost package name (this becomes the <groupId> in Maven and also the '~' value when using Roo's shell); default: '*NULL*' (mandatory)

--projectName

The name of the project (last segment of package name used as default); default: '*NULL*'

--java

Forces a particular major version of Java to be used (will be auto-detected if unspecified; specify 5 or 6 or 7 only); default: '*NULL*'

--parent

The Maven coordinates of the parent POM, in the form "groupId:artifactId:version"; default: '*NULL*'

--packaging

The Maven packaging of this project; default if option not present: 'jar'

Chapter 49. Prop File Commands

Prop File Commands are contained in org.springframework.roo.addon.propfiles.PropFileCommands.

49.1. properties list

Shows the details of a particular properties file

```
properties list --name --path
```

--name

Property file name (including .properties suffix); default: '*NULL*' (mandatory)

--path

Source path to property file; default: '*NULL*' (mandatory)

49.2. properties remove

Removes a particular properties file property

```
properties remove --name --path --key
```

--name

Property file name (including .properties suffix); default: '*NULL*' (mandatory)

--path

Source path to property file; default: '*NULL*' (mandatory)

--key

The property key that should be removed; default: '*NULL*' (mandatory)

49.3. properties set

Changes a particular properties file property

```
properties set --name --path --key --value
```

--name

Property file name (including .properties suffix); default: '*NULL*' (mandatory)

--path

Source path to property file; default: '*NULL*' (mandatory)

--key

The property key that should be changed; default: '*NULL*' (mandatory)

--value

The new vale for this property key; default: '*NULL*' (mandatory)

Chapter 50. Repository Jpa Commands

Repository Jpa Commands are contained in org.springframework.roo.addon.layers.repository.jpa.addon.RepositoryJpaCommands.

50.1. repository jpa

Adds @RooJpaRepository annotation to target type

```
repository jpa --interface
```

--interface

The java interface to apply this annotation to; default: '*NULL*' (mandatory)

--entity

The domain entity this repository should expose; default if option not present: '*'

Chapter 51. Security Commands

Security Commands are contained in org.springframework.roo.addon.security.addon.SecurityCommands.

51.1. permissionEvaluator

Create a permission evaluator

```
permissionEvaluator --package
```

--package

The package to add the permission evaluator to; default: '*NULL*' (mandatory)

51.2. security setup

Install Spring Security into your project

```
security setup
```

This command does not accept any options.

Chapter 52. Selenium Commands

Selenium Commands are contained in org.springframework.roo.addon.web.selenium.SeleniumCommands.

52.1. selenium test

Creates a new Selenium test for a particular controller

```
selenium test --controller
```

--controller

Controller to create a Selenium test for; default: '*NULL*' (mandatory)

--name

Name of the test; default: '*NULL*'

--serverUrl

URL of the server where the web application is available, including protocol, port and hostname; default: 'http://localhost:8080/'

Chapter 53. Service Commands

Service Commands are contained in org.springframework.roo.addon.layers.service.addon.ServiceCommands.

53.1. service all

Adds @RooService annotation to all entities

```
service all --interfacePackage
```

--interfacePackage

The java interface package; default: '*NULL*' (mandatory)

--classPackage

The java package of the implementation classes for the interfaces; default: '*NULL*'

--useXmlConfiguration

When true, Spring Roo will configure services using XML. This is the default behavior for services using GAE; default: '*NULL*'

53.2. service secure all

Adds @RooService annotation to all entities with options for authentication, authorization, and a permission evaluator

```
service secure all --interfacePackage
```

--interfacePackage

The java interface package; default: '*NULL*' (mandatory)

--classPackage

The java package of the implementation classes for the interfaces; default: '*NULL*'

--requireAuthentication

Whether or not users must be authenticated to use the service; default if option present: 'true'; default if option not present: 'false'

--authorizedRole

The role authorized the use the methods in the service (additional roles can be added after creation); default: '*NULL*'

--usePermissionEvaluator

Whether or not to use a PermissionEvaluator; default if option present: 'true'; default if option not present: 'false'

--useXmlConfiguration

When true, Spring Roo will configure services using XML.; default: '*NULL*'

53.3. service secure type

Adds @RooService annotation to target type with options for authentication, authorization, and a permission evaluator

```
service secure type --interface
```

--interface

The java interface to apply this annotation to; default: '*NULL*' (mandatory)

--class

Implementation class for the specified interface; default: '*NULL*'

--entity

The domain entity this service should expose; default if option not present: '*'

--requireAuthentication

Whether or not users must be authenticated to use the service; default if option present: 'ture'; default if option not present: 'false'

--authorizedRoles

The role authorized the use the methods in the service; default: '*NULL*'

--usePermissionEvaluator

Whether or not to use a PermissionEvaluator; default if option present: 'true'; default if option not present: 'false'

--useXmlConfiguration

When true, Spring Roo will configure services using XML.; default: '*NULL*'

53.4. service type

Adds @RooService annotation to target type

```
service type --interface
```

--interface

The java interface to apply this annotation to; default: '*NULL*' (mandatory)

--class

Implementation class for the specified interface; default: '*NULL*'

--entity

The domain entity this service should expose; default if option not present: '*'

--useXmlConfiguration

When true, Spring Roo will configure services using XML.; default: '*NULL*'

Chapter 54. Tailor Commands

Tailor Commands are contained in org.springframework.roo.addon.tailor.TailorCommands.

54.1. tailor activate

Activate a tailor configuration.

```
tailor activate --name
```

--name

The name of the tailor configuration; default: '*NULL*' (mandatory)

54.2. tailor deactivate

Deactivate the tailor.

```
tailor deactivate
```

This command does not accept any options.

54.3. tailor list

List available tailor configurations.

```
tailor list
```

This command does not accept any options.

Chapter 55. Web Finder Commands

Web Finder Commands are contained in org.springframework.roo.addon.web.mvc.controller.addon.finder.WebFinderCommands.

55.1. web mvc finder add

Adds @RooWebFinder annotation to MVC controller type

```
web mvc finder add --formBackingType
```

--formBackingType

The finder-enabled type; default: '*NULL*' (mandatory)

--class

The controller java type to apply this annotation to; default if option not present: '*'

55.2. web mvc finder all

Adds @RooWebFinder annotation to existing MVC controllers

```
web mvc finder all
```

This command does not accept any options.

Chapter 56. Web Json Commands

Web Json Commands are contained in org.springframework.roo.addon.web.mvc.controller.addon.json.WebJsonCommands.

56.1. web mvc json add

Adds @RooJson annotation to target type

```
web mvc json add --jsonObject
```

--jsonObject

The JSON-enabled object which backs this Spring MVC controller.; default: '*NULL*' (mandatory)

--class

The java type to apply this annotation to; default if option not present: '*'

56.2. web mvc json all

Adds or creates MVC controllers annotated with @RooWebJson annotation

```
web mvc json all
```

--package

The package in which new controllers will be placed; default: '*NULL*'

56.3. web mvc json setup

Set up Spring MVC to support JSON

```
web mvc json setup
```

This command does not accept any options.

Appendix B: Usage and Conventions

In this chapter we'll introduce how to use the Roo tool itself. We'll cover typical conventions you'll experience when using Spring Roo.

B.1. Usability Philosophy

As mentioned in earlier chapters and is easily experienced by simply using Spring Roo for a project, we placed a great deal of emphasis on usability during Roo's design. It is our experience that a normal enterprise Java developer is able to pass the ten minute test with Roo and build a new project without referring to documentation. There are several conventions that we use within Roo to ensure a highly usable experience:

- Numerous shell features which ensure the primary Roo-specific user interface is friendly and learnable
- Only using popular, mainstream technologies and standards within Roo applications
- Ensuring Roo works with your choice of IDE or no IDE at all
- Delivering an application architecture that is easy to understand and avoids "magic"
- Making sure Roo works the way a reasonable person would expect it to
- Forgiving mistakes

The last two points are what we're going to discuss in this section.

Making sure Roo works the way you would expect it to is reflected in a number of key design decisions that basically boil down to "you can do whatever you want, whenever you want, and Roo will automatically work in with you". There are obviously limits to how far we can take this, but as you use Roo you'll notice a few operational conventions that underpin this.

Let's start by looking at file conventions. Roo will never change a `.java` file in your project unless you explicitly ask it to via a shell command. In particular, Roo will not modify a `.java` file just because you apply an annotation. Roo also handles .xml files in the same manner. There are only two file types that may be created, updated or deleted by Roo in an automatic manner, those being `.jspx` files and also AspectJ files which match the `*_Roo_*.aj` wildcard.

In terms of the AspectJ files, Roo operates in a very specific manner. A given AspectJ filename indicates the "target type" the members will be introduced into and also the add-on which governs the file. Roo will only ever permit a given AspectJ file to be preserved if the target type exists and the corresponding add-on requests an ITD for that target type. Nearly all add-ons will only create an ITD if there is a "trigger annotation" on the target type, with the trigger annotation always incorporating an @Roo prefix. As such, if you never put any @Roo annotation on a given .java file, you can be assured Roo will never create any AspectJ ITD for that target type. Refer to the file system conventions section for

related information.

You'll also notice when using Roo that it automatically responds to changes you make outside Roo. This is achieved by an auto-scaling file system monitoring mechanism. This basically allows you to create, edit or delete any file within your project and if the Roo shell is running it will immediately detect your change and take the necessary action in response. This is how round-tripping works without you needing to include Roo as part of your build system or undertake any crude mass generation steps.

What happens if the Roo shell isn't running? Will there be a problem if you forget to load it and make a change? No. When Roo starts up it performs a full scan of your full project file system and ensures every automatically-managed file that should be created, updated or deleted is handled accordingly. This includes a full in-memory rebuild of each file, and a comparison with the file on disk to detect changes. This results in a lot more robust approach than relying on relatively coarsely-grained file system timestamp models. It also explains why if you have a very big project it can take a few moments for the Roo shell to startup, as there is no alternative but to complete this check for actions that happened when Roo wasn't running.

The automated startup-time scan is also very useful as you upgrade to newer versions of Roo. Often a new version of Roo will incorporate enhancements to the add-ons that generate files in your project. The startup-time scan will therefore automatically deliver improvements to all generated files. This is also why you cannot edit files that Roo is responsible for managing, because Roo will simply consider your changes as some "old format" of the file and rewrite the file in accordance with its current add-ons.

Not being able to edit the generated files may sound restrictive, as often you'll want to fine-tune just some part of the file that Roo has emitted. In this case you can either write a Roo add-on, or more commonly just write the method (or field or constructor etc) directly in your .java file. Roo has a convention of detecting if any member it intends to introduce already exists in the target type, and if it does Roo will not permit the ITD to include that member. In plain English that means if you write a method that Roo was writing, Roo will remove the method from its generated file automatically and without needing an explicit directive to do so. In fact the Roo core infrastructure explicitly detects buggy add-ons that are trying to introduce members that an end user has written and it will throw an exception to prevent the add-on from doing so.

This talk of exceptions also lets us cover the related usability feature of being forgiving. Every time Roo changes your file system or receives a shell command, it is executed within a quasi-transactional context that supports rollback. As a result, if anything goes wrong (such as you made a mistake when entering a command or an add-on has a problem for whatever reason) the file system will automatically rollback to the state it was before the change was attempted. The cascading nature of many changes (i.e. you add a field to a `.java` file and that changes an AspectJ ITD and that in turn changes a web `.jspx` etc) is handled in the same unit of work and therefore rolled back as an atomic group when required.

Before leaving this discussion on usability, it's probably worth pointing out that although the Roo shell contains numerous commands, you don't need to use them. You are perfectly free to perform any change to your file system by hand (without the help of the Roo shell). For example, there are

commands which let you create `.java` files or add fields to them. You can use these commands or you can simply do this within your IDE or text editor. Roo's automatic file system monitoring will detect the changes and respond accordingly. Just work the way you feel most comfortable - Roo will respect it.

B.2. Shell Features

Many people who first look at Roo love the shell. In fact when we first showed Roo to an internal audience, one of the developers present said tounge-in-cheek, "That could only have come from someone with a deep love of the Linux command line!". All jokes aside, the shell is only one part of the Roo usability story - although it's a very important part. Here are some of the usability features that make the shell so nice to work with:

- *Tab completion*: The cornerstone of command-line usability is tab assist. Hit TAB (or CTRL+SPACE if you're in SpringSource Tool Suite) and Roo will show you the applicable options.

- *Command hiding*: Command hiding will remove commands which do not make sense given the current context of your project. For example, if you're in an empty directory, you can type project, hit TAB, and see the options for creating a project. But once you've created the project, the project command is no longer visible. The same applies for most Roo commands. This is nice as it means you only see commands which you can actually use right now. Of course, a full list of commands applicable to your version of Roo is available in the command index appendix and also via help.

- *Contextual awareness*: Roo remembers the last Java type you are working with in your current shell session and automatically treats it as the argument to a command. You always know what Roo considers the current context because the shell prompt will indicate this just before it writes **roo**>. In the command index you might find some options which have a default value of '*'. This is the marker which indicates "the current context will be used for this command option unless you explicitly specify otherwise". You change the context by simply working with a different Java type (i.e. specify an operation that involves a different Java type and the context will change to that Java type).

- *Hinting*: Not sure what to do next? Just use the hint command. It's the perfect lightweight substitute for documentation if you're in a hurry!

- *Inbuilt help*: If you'd like to know all the options available for a given command, use the help command. It lists every option directly within the shell.

- *Automatic inline help*: Of course, it's a bit of a pain to have to go to the trouble of typing help then hitting enter if you're in the middle of typing a command. That's why we offer inline help, which is automatically displayed whenever you press TAB. It is listed just before the completion options. To save screen space, we only list the inline help once for a given command option. So if you type `project --template TAB TAB TAB`, the first time you press TAB you'd see the inline help and the completion options

- *Scripting and script recording*: Save your Roo commands and play them again later.

- *Prevent changes if Roo version changes*: Since Spring Roo 2.0, roo shell prevents automatic changes if detects that generated project was started with a different version of Spring Roo shell.

The scripting and script recording features are particularly nice, because they let you execute a series of Roo commands without typing them in.

To execute a Roo script, just use the script command. When you use the script command you'll need to indicate the script to run. We ship a number of sample scripts with Roo, as discussed earlier in the Exploring Roo Samples section.

What if you want to create your own scripts? All you need is a text editor. The syntax of the script is identical to what you'd type at the Roo shell. Both the Roo shell and your scripts can contain inline comments using the `;` and `//` markers, as well as block comments using the `/* */` syntax.

A really nice script-related feature of the Roo shell is that it will automatically build a script containing the commands you entered. This file is named `log.roo` and exists in your current working directory. Here's a quick example of the contents:

```
// Spring Roo ENGINEERING BUILD [rev 553:554M] log opened at 2009-12-31 08:10:58
project setup --topLevelPackage roo.shell.is.neat
// [failed] jpa setup --database DELIBERATE_ERROR --provider HIBERNATE
jpa setup --database HYPERSONIC_IN_MEMORY --provider HIBERNATE
quit
// Spring Roo ENGINEERING BUILD [rev 553:554M] log closed at 2009-12-31 08:11:37
```

In the recorded script, you can see the version number, session start time and session close times are all listed. Also listed is a command I typed that was intentionally incorrect, and Roo has turned that command into a comment within the script (prefixed with `// [failed]`) so that I can identify it and it will not execute should I run the script again later. This is a great way of reviewing what you've done with Roo, and sharing the results with others.

B.3. IDE Usage

Despite Roo's really nice shell, in reality most people develop most of their application using an IDE or at least text editor. Roo fully expects this usage and supports it.

Before we cover how to use an IDE, it's worth mentioning that you don't strictly need one. With Roo you can build an application at the command line, although to be honest you'll get more productivity via an IDE if it's anything beyond a trivial application. If you would prefer to use the command line, you can start a fresh application using the Roo shell, edit your `.java` and other files using any text editor, and use the perform commands to compile, test and package your application ready for deployment. You can even use `mvn tomcat:run` to execute a servlet container. Again, you'll be more productive in an IDE, but it's nice to know Roo doesn't force you to use an IDE unless you'd like to use one.

In relation to IDEs, we highly recommend that you use SpringSource Tool Suite (STS). STS is a significantly extended version (and free!) of the pervasive Eclipse IDE. From a Roo perspective, STS preintegrates the latest AspectJ Development Tools (AJDT) and also offers Roo shell as extension plugin.

The STS Roo shell means you do not need to run the normal Roo shell if you are using STS. You'll also have other neat Roo-IDE integation features, like the ability to press CTRL+R (or Apple+R if you're on an Apple) and a popup will allow you to type a Roo command from anywhere within the IDE. Another nice feature is the shell message hotlinking, which means all shell messages emitted by Roo are actually links that you can click to open the corresponding file in an Eclipse editor.

Naturally Roo works well with standard Eclipse as well. All you need to do is ensure you install the latest AspectJ Development Tools (AJDT) plugin. This will ensure code assist and incremental compilation works well. We also recommend you go into Window > Preferences > General > Workspace and switch on the "Refresh automatically" box. That way Eclipse will detect changes made to the file system by the externally-running Roo shell. It's also recommended to install the m2eclipse plugin, which is automatically included if you use STS and is particularly suitable for Roo-based projects.

When using AJDT you may encounter a configuration option enabling you to "weave" the JDT. This is on by default in STS, so you're unlikely to see the message if using STS. If you are prompted (or locate the configuration settings yourself under the Window > Preferences > JDT Weaving menu), you should enable weaving. This ensures the Java Editor in Eclipse (or STS) gives the best AspectJ-based experience, such as code assist etc. You can also verify this setting is active by loading Eclipse (or STS) and selecting Window > Preferences > JDT Weaving.

If you're using m2eclipse, you won't need to use the perform eclipse command to setup your environment. A simple import of the project using Eclipse's File > Import > General > Maven Projects menu option is sufficient.

Irrespective of how you import your project into Eclipse (i.e. via the perform eclipse command or via m2eclipse) you should be aware that the project will not be a Web Tools Project (WTP) until such time as you install your first web controller. This is usually undertaken via the web mvc all or web mvc controller command. If you have already imported your project into Eclipse, simply complete the relevant *web mvc* command and then re-import. The project will then be a WTP and offer the ability to deploy to an IDE-embedded web container. If you attempt to start a WTP server and receive an error message, try right-clicking the project and selecting Maven > Update Project Configuration. This often resolves the issue.

If you're using IntelliJ, we are pleased to report that IntelliJ now supports Roo. This follows the completion of ticket IDEA-26959, where you can obtain more information about the AspectJ support now available in IntelliJ.

If you're using any IDE other than STS, the recommended operating pattern is to load the standalone Roo shell in one operating system window and leave it running while you interact with your IDE. There is no formal link between the IDE and Roo shell. The only way they "talk" to each other is by both monitoring the file system for changes made by the other. This happens so quickly that you're

unlikely to notice, and indeed internally to Roo we have an API that allows the polling-based approach to be replaced with a formal notification API should it ever become necessary. As discussed in the usability section, if you forget to load the Roo shell and start modifying your project anyway, all you need to do is load the Roo shell again and it will detect any changes it needs to make automatically.

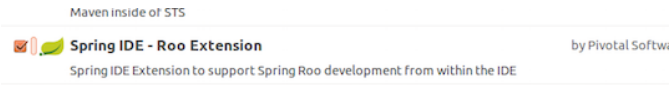
B.3.1. Install the STS Roo Support

STS Roo Support for Spring Roo 2.0.0 is available in STS 3.7.0 and later only. Go to Spring Tool Suite™ Downloads and follow the instructions.

Now download Spring Roo 2.0.0 from Spring Roo project page and unzip the distribution.

To include Roo on your STS follow the instructions below:

1. Open your STS IDE.
2. Open STS dashboard.
3. Click on Extensions bottom tab and search Spring Roo.
4. Install Spring IDE - Roo Extension.

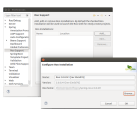


5. Restart STS IDE

Configure Spring Roo 2.0.0 on your STS:

1. Open "Preferences > Spring > Roo Support".
2. In "Roo Support" press "Add" new installation button.
3. In "Roo Configure Roo Installation" press "Browse" button, then select the the directory in which Spring Roo 2.0.0 was unpacked.

4. Confirm the new Roo installation.



5. Now Roo is installed in your STS.

B.4. Build System Usage

Roo currently supports the use of Apache Maven. This is a common build system used in many enteprise applications. We routinely poll our community and look at public surveys which consistently show that nearly all enterprise development projects use either Maven or Ant, so we believe this is a good default for Roo projects. As per the installation instructions, you must ensure you are using Maven 3.0.5 or above. We do recommend you use last version of Maven for best results, though.

Roo will create a new `pom.xml` file whenever you use the project command. The POM will contain the following Roo-specific considerations:

- A reference to the Roo annotations JAR. This JAR exists at development time only and has a scope that prevents it from being included in resultant WAR files.
- A correct configuration of the Maven AspectJ plugin. This includes a reference to the Spring Aspects library, which is important to Roo-based applications. Spring Aspects is included within Spring Framework.

There are no other Roo changes to the POM. In particular, there is no requirement for the POM to include Roo as part of any code generation step. Roo is never used in this "bulk generation style".

If you are interested in ensuring a build includes the latest Roo code generation output, you can cause Maven or equivalent build system to execute `roo quit`. The presentation of the quit command line option will cause the Roo shell to load, perform its startup-time scan (which identifies and completes any required changes to generated files) and then exit.

Those seeking Ant/Ivy instead of Maven support are encouraged to vote for issue ROO-91. The internals of Roo do not rely on Maven at all. Nonetheless we have deferred it until we see sufficient community interest to justify maintaining two build system environments.

B.5. File System Conventions

We have already covered some of Roo's file system conventions in the Usability Philosophy section. In summary Roo will automatically monitor the file system for changes and code generate only those files which match the `*_Roo_*.aj` wildcard. It will also code generate those JSPs associated with scaffolded MVC controllers that have the annotation `@RooWebScaffold`.

Roo applications follow the standard Maven-based directory layout. We have also placed Spring application context-related files (both `.xml` and `.properties`) in the recommended classpath sub-directory for Spring applications, `META-INF/spring`.

B.6. Add-On Installation and Removal

Roo supports the installation and removal of third-party add-ons. Roo added significant enhancements to its add-on model, as more thoroughly discussed in Part III of this manual.

B.7. Recommended Practices

Following some simple recommendations will ensure you have the best possible experience with Roo:

- Don't edit any files that Roo code generates (see the Usability Philosophy for details).
- Before installing any new technology, check if Roo offers a setup command and use it if present (this will ensure the seutp reflects our recommendations and the expectations of other add-ons).
- Ensure you leave the Roo shell running when creating, updating or deleting files in your project.
- Remember you'll still need to write Java code (and JSPs for custom controllers). Have the right expectations before you start using Roo. It just helps you - it doesn't replace the requirement to program.
- Check the Known Issues section before upgrading or if you experience any problems.
- Refer to the Roo Resources section for details of how to get assistance with Roo, such as the forum and issue tracking database. We're happy to hear from you.

Appendix C: Existing Building Blocks

Sometimes you have an existing project or database. This chapter covers how to make Spring Roo work with it.

C.1. Existing Projects

If you have an existing project that you'd like to use with Roo, we recommend that you follow these steps:

1. Decide whether your project files are easier to migrate to a new Roo project or it's easier to amend your current project into a Roo project. Both approaches are valid. The following steps reflect migrating your current project into a Roo project.
2. Convert the project to use Maven. Ensure you use the correct Maven directory layouts.
3. Move your Spring configuration and other files to the same directories as used by Roo. Start a new Roo-based project if you're unsure where these files are typically stored.
4. Add the Roo annotations JAR and Maven AspectJ plugin to your POM. Use the same syntax as a new Roo-based project would use.
5. Load Roo on your project and verify it does not report any errors. Resolve any errors before continuing.
6. Add a test @RooToString annotation to one of your existing classes. Verify the ITD is created and can be used within your IDE (if you're using an IDE). Check the new toString() method is used.
7. Start incrementally using the simpler Roo add-ons like toString support and JavaBeans. When you're confident, move onto other Roo commands and add-ons.

If you encounter any difficulty, we recommend you consult the Roo Resources section of the reference guide for help.

C.2. Existing Databases

Many organisations have existing databases that they'd like to use with Roo.

A significant new feature added to Spring Roo 1.1 was support for incremental database reverse engineering. This feature is robust and comprehensive, and allows you to reverse engineer an existing database in a single command. The single command doesn't even ask you any questions as it operates, and it gracefully handles changes to your schema over time.

We recommend that you consult the incremental database reverse engineering chapter if you'd like to work with an existing relational database.

Appendix D: Removing Roo

While we'll be sad to see you go, we're happy that Roo was able to help you in some way with your Spring-based projects. We also know that most people reading this chapter aren't actually likely to remove Roo at all, and are simply wondering how they'd go about it in the unlikely event they ever actually wanted to. If you have a source control system, it's actually a good idea to complete these instructions (without checking in the result!) just to satisfy yourself that it's very easy and reliable to remove Roo.

D.1. How Roo Avoids Lock-In

At the time we created the mission statement for Roo, a key dimension was "*without compromising engineering integrity or flexibility*". To us that meant not imposing an unacceptable burden on projects like forcing them to use the Roo API or runtime or locking them in. While it complicated our design to achieve this, we're very proud of the fact Roo's approach has no downside at runtime or lock-in or future flexibility. You really can have your cake and eat it too, to reflect on the common English expression.

Roo avoids locking you in by adopting an active code generation approach, but unlike other code generators, we place Roo generated code in separate compilation units that use AspectJ inter-type declarations. This is vastly better than traditional active code generation alternatives like forcing you to extend a particular class, having the code generator extend one of your classes, or forcing you to program a model in an unnatural diagrammatic abstraction. With Roo you just get on with writing Java code and let Roo take care of writing and maintaining the code you don't want to bother writing.

The other aspect of how Roo avoids lock-in is using annotations with source-level retention. What this means is the annotations are not preserved in your `.class` files by the time they are compiled. This in turn means you do not need the Roo annotation library in your runtime classpath. If you look at your `WEB-INF/lib `directory (if you're building a web project), you will find absolutely no Roo-related JARs. They simply don't exist. In fact if you look at your development-time classpath, only the Roo annotation JAR library will be present - and that JAR doesn't contain a single executable line of code. The entire behaviour of Roo is accomplished at development time when you load the Roo shell. If you also think about the absence of executable code anywhere in your project classpath, there is no scope for possible Roo bugs to affect your project, and there is no risk of upgrading to a later version of Roo.

Because we recommend people check their Roo-generated `*_Roo_*.aj` files into source control, you don't even need to load Roo to perform a build of your project. The source-level annotation library referred to in the previous paragraph is in a public Maven repository and will automatically be downloaded to your computer if it's not already present. This means Roo is not part of your build process and your normal source control system branching and tagging processes will work.

This also means that a project can "stop using Roo" by simply never loading the Roo shell again. Because the `*_Roo_*.aj` files are written to disk by the Roo shell when it last ran, even if it's never loaded again those files will still be present. The removal procedures in this chapter therefore focus on

a more complete removal, in that you no longer even want the `*_Roo_*.aj` files any more. That said, there's nothing wrong with just never loading Roo again and keeping the `*_Roo_*.aj` files. The only possible problem of adopting the "never load Roo again" approach is that someone might load Roo again and those files will be updated to reflect the latest optimisations that Roo can provide for you.

D.2. Pros and Cons of Removing Roo

By removing Roo, you eliminate the Roo-generated source files from your project. These are inter-type declarations stored in `*_Roo_*.aj` files. You also remove the Roo annotation library from your project. This might be attractive if you've made a decision to no longer use Roo for some reason, or you'd like to ship the finished project to your client and they'd prefer a simple Java project where every piece of code is in standard `.java` files. Another reason you might like to remove Roo is to simply satisfy yourself it's easy to do so and therefore eliminate a barrier to adopting Roo for real projects in the first place.

Even though it's easy to do so, there are downsides of removing Roo from your project:

- *Cluttered Java classes*: If the `*_Roo_*.aj` files are removed, their contents need to go somewhere. That somewhere is into your `.java` source files. This means your `.java` source files will be considerably longer and contain code that no developer actually wrote. When developers open your `.java` source files, they'll need to figure out what was written by hand and is unique to the class, what was automatically generated and then modified, and what was automatically generated and never modified. If using Roo this problem is eliminated, as anything automatically generated is in a separate, easily-identified source file.

- *No round-trip support*: Let's imagine for a moment that you've written (either by hand or via your IDE's code generation feature) a `toString()` method and getter/setter pairs for all your fields. You then decide to rename a field. Suddenly the getter, setter and `toString()` methods are all in error. If you use Roo, it automatically detects your change and appropriately updates the generated code. If you remove Roo, you'll lose this valuable round-trip support and be doing a lot more tedious work by hand.

- *No optimisations to generated files*: With each version of Roo we make improvements to the automatically-created `*_Roo_*.aj` files. These improvements are automatically made to your `*_Roo_*.aj` files when you load a new version of Roo. These improvements occasionally fix bugs, but more often provide new features and implement existing features more efficiently (remember eliminating engineering trade-offs and therefore maximising efficiency is a major objective in our mission statement). If you remove the `*_Roo_*.aj` files, you'll receive the code as of that date and you'll miss out on further improvements we make.

- *Loss of Roo commands*: There are dozens of Roo commands available to assist you adapt to evolving project requirements. Next month you might be asked to add JMS services to your project. With Roo you just "jms setup". The month after you're asked about SMTP, so you just "email sender setup". If you've eliminated Roo, you'll need to resort to much more time-consuming manual configuration (with its associated trial and error).

- *Deprecated library versions*: Because Roo automatically updates your code and has a good knowledge of your project, it's easy to always use the latest released versions of important runtime technologies like Spring and JPA. If you stop using Roo, you'll need to manually do all of the work involved in upgrading your project to newer versions. This will mean you're likely to end up on older runtime library versions that have bugs, fewer features and are not maintained or supported. With Roo you significantly mitigate this risk.

- *Undesirable architectural outcomes*: With Roo you achieve team-wide consistency and a solution with a high level of engineering integrity. If developers are forced to write repetitious code themselves and no longer enjoy optimised Roo commands, you'll likely find that over time you lose some of the consistency and engineering advantages of having used Roo in the first place.

- *Higher cost*: With the above in mind, you'll probably find development takes longer, maintenance takes longer and your runtime solution will be less efficient than if you'd stayed with Roo.

As such we believe using Roo and continuing to use Roo makes a lot of sense. But if you're willing to accept the trade-offs of removing Roo (which basically means you switch to writing your project the unproductive "old fashioned way"), you can remove Roo very easily. Don't forget when in doubt you can always defer the decision. It's not as if Roo won't let you remove it just as easily in six months or two years from now!

D.3. Step-by-Step Removal Instructions

The following instructions explain how to remove Spring Roo from one of your projects that has to date been using Roo. Naturally if you'd simply like to remove Roo from your computer (as opposed to from an existing project), the process is as simple as removing the Roo installation directory and symbolic link. This section instead focuses on the removal from your projects.

As mentioned above, a simple way of stopping to use Roo is to simply never load it again. The `*_Roo_*.aj` files will still be on disk and your project will continue to work regardless of whether the Roo shell is never launched again. You can even uninstall the Roo system from your computer and your project will still work. The advantage of this approach is you haven't lost most of the benefits of using Roo and it's very easy to simply reload the Roo shell again in the future. This section covers the more complete removal option should you not even want the `*_Roo_*.aj` files any more.

Please be aware that enhancement request ROO-222 exists to replace step 1 with a Roo command, and ROO-330 similarly focuses on steps 2 and 3. Please vote for these enhancement requests if you'd like them actioned, although the instructions below still provide a fast and usable removal procedure.

D.3.1. Step 1: Push-In Refactor

Before proceeding, ensure you have quit any running Roo shell. We also recommend you run any tests and load your web application interface (if there is one) to verify your project works correctly before starting this procedure. We also recommend that you create a branch or tag in your source control repository that represents the present "Roo-inclusive" version, as it will help you should you ever wish

to reenable Roo after a removal.

To remove Roo from a project, you need to import the project into Eclipse or SpringSource Tool Suite. Once the project has been imported into Eclipse, right-click the project name in Package Explorer and select Refactor > Push-In Refactor. If this option is missing, ensure that you have a recent version of AJDT installed. After selecting the push-in refactor menu option, a list of all Roo inter-type declarations will be displayed. Simply click OK. AJDT will have now moved all of the Roo inter-type declarations into your standard `.java` files. The old `*_Roo_*.aj` files will have automatically been deleted.

D.3.2. Step 2: Annotation Source Code Removal

While your project is now free of inter-type declarations, your `.java` files will still have `@Roo` annotations within them. In addition, there will be `import` directives at the top of your `.java` files to import those `@Roo` annotations. You can easily remove these unwanted members by clicking Search > Search > File Search, containing text "`\n.*[@\.]Roo[^t_]+?.*$`" (without the quotes), file name pattern "`*.java`" (without the quotes), ticking the "Regular expression" and "Case sensitive" check-boxes and clicking "Replace". When the next window appears and asks you for a replacement pattern, leave it blank and continue. All of the Roo statements will have now been removed. We have noticed for an unknown reason that sometimes this operation needs to be repeated twice in Eclipse.

D.3.3. Step 3: Annotation JAR Removal

By now your `.java` files do not contain any Roo references at all. You therefore don't require the `org.springframework.roo.annotations-*.jar` library in your development-time classpath. Simply open your `pom.xml` and locate the `<dependency>` element which contains `<artifactId>org.springframework.roo.annotations</artifactId>`. Delete (or comment out) the entire `<dependency>` element. If you're running m2Eclipse, there is no need to do anything further. If you used the command-line `mvn` command to create your Eclipse `.classpath` file, you'll need to execute `mvn eclipse:clean eclipse:eclipse` to rebuild the `.classpath` file.

Roo has now been entirely removed from your project and you should re-run your tests and user interface for verification of expected operation. It's probably a good idea to perform another branch or tag in your source control repository so the change set is documented.

D.4. Reenabling Roo After A Removal

If you decide to change your mind and start using Roo again, the good news is that it's relatively easy. This is because your project already uses the correct directory layout and has AspectJ etc properly configured. To re-enable Roo, simply open your pom.xml and re-add the `org.springframework.roo.annotations` `<dependency>` element. You can obtain the correct syntax by simply making a new directory, changing into that directory, executing `roo script vote.roo`, and inspecting the resulting `pom.xml`.

Once you've added the dependency, you're free to load Roo from within your project's directory and start using the Roo commands again. You're also free to add `@Roo` annotations to any `.java` file that

would benefit from them, but remember that Roo is "hands off by default". What that means is if you used the push-in refactor command to move members (e.g. fields, methods, annotations etc) into the `.java` file, Roo has no way of knowing that they originated from a push-in refactor as opposed to you having written them by hand. Roo therefore won't delete any members from your `.java` file or override them in an inter-type declaration.

Our advice is therefore (a) don't remove Roo in the first place or (b) if you have removed Roo and go back to using Roo again, delete the members from your `.java` files that Roo is able to automatically manage for you. By deleting the members that Roo can manage for you from the `.java` files, you'll gain the maximum benefit of your decision to resume using Roo. If you're unsure which members Roo can automatically manage, simply comment them out and see if Roo provides them automatically for you. Naturally you'll need the relevant `@Roo` annotation(s) in your `.java` files before Roo will create any members automatically for you.

A final tip if you'd like to return to having ITDs again is that AJDT 2.0 and above offers a Refactor > Push Out command. This may assist you in moving back to ITDs. The Edit > Undo command also generally works if you decide to revert immediately after a Refactor > Push In operation.

Appendix E: Upgrade Notes and Known Issues

E.1. Known Issues

Because Spring Roo integrates a large number of other technologies, invariably some people using Roo may experience issues when using certain combinations of technologies together. This section aims to list such known issues in an effort to help you avoid experiencing any problems. If you are able to contribute further information, a solution or workaround to any of these known issues, we'd certainly appreciate hearing from you via the community forums.

- *JDK compatibility*: Spring Roo has been tested with Sun, IBM, JRockit and Apache Harmony JVMs for Java 5 and Java 6. We do not formally support other JVMs or other versions of JVMs. We have also had an issue reported with versions of Java 6 before 1.6.0_17 due to Java bug 6506304 and therefore recommend you always use the latest released version of Java 6 for your platform. There is also a known issue with OpenJDK. You can read about our testing of different JDKs in issue ROO-106.

- *Human language support*: Pluralisation within Roo delegates to the Inflector library. Due to some issues with Inflector, only English pluralisation is supported. If you wish to override the plural selected by Inflector (and in turn used by Roo), you can specify a particular plural for either a Java type or Java field by using the `@RooPlural` annotation. Longer term it would be nice if someone ported the Inflector code into the Roo pluralisation add-on so that we can fix these issues and support other languages. We are receptive to contributions from the community along these lines.

- *Shell wrapping*: In certain cases typing a long command into the shell that wraps over a single line may prevent you from being able to backspace to the prior line. This is caused by the JLine library (not Roo). We expect to rewrite the shell at some future time and will likely stop using JLine at that point.

- *Hibernate issues*: Hibernate is one of the JPA providers we test with, however, Hibernate has issues with `--mappedSuperclass` as detailed in ROO-292 and ROO-747. We recommend you do not use `--mappedSuperclass` in combination with Hibernate. We have found OpenJPA works reliably in all cases, so you might want to consider switching to OpenJPA if you are seriously impacted by this issue (the "jpa setup" command can be used multiple times, which is useful for experimentally switching between different JPA providers).

- *Integration testing limitations*: The data on demand mechanism (which is used for integration tests) has limited JSR 303 (Bean Validator) compatibility. Roo supports fields using @NotNull, @Past and @Future, @Size, @Min, and @Max. No other validator annotations are formally supported, although many will work. To use other validator annotations, you may need to edit your DataOnDemand.java file and add a manual `getNewTransientEntity(int)` method. Refer to a generated `*_Roo_DataOnDemand.aj` file for an example. Alternately, do not use the integration test functionality in Roo unless you have relatively simple validation constraints or you are willing to provide this data on demand method.

- *Tomcat 5.5*: Tomcat 5.5 can not be supported by the scaffolded Spring MVC Web UI. Tomcat 5.5 does

not support the JSP 2.1 API. Roo makes extensive use of the JSP 2.1 API in the scaffolded Web UI (specifically expression language features). Furthermore, the JSP 2.0 API does not support JDK 5 enums (a feature that Roo would need). See ROO-680 for more details. The following forum post offers a workaround for the JSP 2.1 incompatibility issue. Please be aware that this has not been tested by the Roo team and Tomcat 5.5 does officially support the JSP 2.0 API.

- Applications with a scaffolded Spring MVC UI are currently not deployable to Google App Engine due to incompatibilities in the JSP support and JSTL. See ROO-1006 for details.
- Applications with a scaffolded GWT UI require a manual adjustment in `src/main/webapp/WEB-INF/spring/webmvc-config.xml` (this will not be required when using Spring Framework 3.0.5+):

```
<mvc:default-servlet-handler <strong>default-servlet-name="_ah_default"</strong> />
```

E.2. Version Numbering Approach

Spring Roo observes version number standards based on the Apache Portable Runtime (APR) versioning guidelines as well as the OSGi specifications. In summary this means all Roo releases adopt the format of MAJOR.MINOR.PATCH.TYPE. Each segment is separated by a period without any spaces. The MAJOR.MINOR.PATCH are always integer numbers, and TYPE is an alphanumeric value. For example, Roo 1.0.3.M1 means major version 1, minor version 0, patch number 3 and release type M1.

You can always rely on the natural sort order of the version numbers to arrive at the latest available version. For example, 1.0.4.RELEASE is more recent than 1.0.4.RC2. This is because "RELEASE" sorts alphabetically lower than "RC2". The TYPE segment can generally be broken into two further undelimited portions, being the release type and a numeric increment. For example, RC1 means release candidate 1 and RC4 means release candidate 4. One exception to this is RELEASE means the final general availability of that release. Other common release types include "A" for alpha and "M" for milestone.

We make no guarantees regarding the compatibility of any release that has a TYPE other than "RELEASE". However, for "RELEASE" releases we aim to ensure you can use a given "RELEASE" with any other "RELEASE" which has the same MAJOR.MINOR version number. As such you should be able to switch from 1.0.4.RELEASE to 1.0.9.RELEASE without any changes. However, you might have trouble with 1.0.4.RELEASE to 1.0.9.RC1 as RC1 is a work-in-progress and we may not have identified all regression issues. Obviously this version portability is only our objective, and sometimes we need to make exceptions or may inadvertently overlook an issue. We appreciate you logging a bug report if you identify a version regression that violates the conventions expressed in this section, so that at least we can confirm it and either attempt to remedy it on the next release of that MAJOR.MINOR version range or bring it to people's attention in the other sections of this appendix.

When upgrading you should review the issue tracker for what has changed since the last version. Because most releases include a large number of issues in the release notes, we attempt to highlight any major issues that may require your attention in the sections below. These notes are not all-

encompassing but simply pointers to the main upgrade-related issues that most people should be aware of. They are also written assuming you are maintaining currency with the latest public releases of Spring Roo and therefore the changes you may need to make to your project are cumulative.

E.3. Upgrading To Any New Release

Before upgrading any project to the next release of Spring Roo, you should always:

- Run the backup command using your currently-installed (i.e. existing) version of Spring Roo. This will help create a ZIP of your project, which may help if you need to revert. Don't install the new version of Roo until you've firstly completed this backup. Naturally you can skip this step if you have an alternate backup technique and have confidence in it.

Spring Roo will update your project's `pom.xml` versions automatically on your behalf when you load it on an existing project.

If you experience any difficulty with upgrading your projects, please use the community support forum for assistance.

Appendix F: Project Background

This chapter briefly covers the history of the Spring Roo project, and also explains its mission statement in detail.

F.1. History

The Spring Roo available today is the result of relatively recent engineering, but the inspiration for the project can be found several years earlier.

The historical motivation for "ROO" can be traced back to 2005. At that time the project's founder, Ben Alex, was working on several enterprise applications and had noticed he was repeating the same steps time and time again. Back in 2005 it was common to use a traditional layering involving DAOs, services layer and web tier. A good deal of attention was also focused around that time on avoiding anaemic domain objects and instead pursuing Domain Driven Design principles.

Pursuing a rich domain model led to domain objects that reflected proper object oriented principles, such as careful application of encapsulation, immutability and properly defining the role of domain objects within the enterprise application layering. Rich behaviour was added to these entities via AspectJ and Spring Framework's recently-created @Configurable annotation (which enabled dependency injection on entities irrespective of how the entities were instantiated). Naturally the web frameworks of the era didn't work well with these rich domain objects (due to the lack of accessors, mutators and no-argument constructors), and as such data transfer objects (DTOs) were created. The mapping between DTOs and domain objects was approached with assembly technologies like Dozer. To make all of this work nicely together, a code generator called Real Object Oriented - or "ROO" - was created. The Real Object Oriented name reflected the rich domain object principles that underpinned the productivity tool.

ROO was presented to audiences at the SpringOne Americas 2006 and TSSJS Europe 2007 conferences, plus the Stockholm Spring User Group and Enterprise Java Association of Australia. The audiences were enthusiastic about the highly productive solution, with remarks like "it is the really neatest and newest stuff I've seen in this conference" and "if ROO ever becomes an open source project, I'm guessing it will be very polished and well-received". Nonetheless, other priorities (like the existing Spring Security project) prevented the code from becoming release-ready. More than twelve months later Ben was still regularly being asked by people, "whatever happened to the ROO framework?" and as such he set out about resuming the project around August 2008.

By October 2008 a large amount of research and development had been undertaken on the new-and-improved ROO. The original productivity ideas within ROO had been augmented with considerable feedback from real-life use of ROO and the earlier conferences. In particular a number of projects in Australia had used the unreleased ROO technology and these projects provided a great deal of especially useful feedback. It was recognised from this feedback that the original ROO model suffered from two main problems. First, it did not provide a highly usable interface and as such developers required a reasonable amount of training to fully make use of Roo. Second, it imposed a high level of

architectural purity on all applications - such as the forced use of DTOs - and many people simply didn't want such purity. While there were valid engineering reasons to pursue such an architecture, it was the productivity that motivated people to use ROO and they found the added burden of issues like DTO mapping cancelled out some of the gains that ROO provided. A mission statement was drafted that concisely reflected the vision of the project, and this was used to guide the technical design.

In early December 2008 Ben took a completely rewritten ROO with him to SpringOne Americas 2008 and showed it to a number of SpringSource colleagues and community members. The response was overwhelming. Not only had the earlier feedback been addressed, but many new ideas had been incorporated into the Java-only framework. Furthermore, recent improvements to AspectJ and Spring had made the entire solution far more effective and efficient than the earlier ROO model (such as annotation-based component scanning, considerable enhancements to AJDT etc).

Feedback following the December 2008 demonstrations led to considerable focus on bringing the ROO technology to the open source community. The name "ROO" was preserved as a temporary codename, given that we planned to select a final name closer to official release. The "ROO" project was then publicly presented on 27 April 2009 during Rod Johnson's SpringOne Europe keynote, "The Future of Java Innovation". As part of the keynote the ROO system was used to build a voting application that would allow the community to select a final name for the new project. The "ROO" name was left as an option, although the case was changed to "Roo" to reflect the fact it no longer represented any acronym. The resulting votes were Spring Roo (467), Spring Boost (180), Spring Spark (179), Spring HyperDrive (64) and Spring Dart (62). As such "Spring Roo" became the official, community-selected name for the project.

Roo 1.0.0.A1 was released during the SpringOne Europe 2009 conference, along with initial tooling for SpringSource Tool Suite. The Roo talk at the SpringOne Europe 2009 conference was the most highly attended session and there was enormous enthusiasm for the solution. Roo 1.0.0.A2 was published a few weeks later, followed by several milestones. By SpringOne/2GX North America in October 2009, Roo 1.0.0 had reached Release Candidate 2 stage, and again the Roo session was the most highly attended session of the entire conference. SpringSource also started hosting the highly popular Spring Discovery Days and showing people around the world what they could do with the exciting new Roo tool. Coupled with Twitter, by this stage many members of the Java community had caught a glimpse of Roo and it was starting to appear in a large number of conferences, user group meetings and development projects - all before it had even reached 1.0.0 General Availability!

Spring Roo 1.1.0 was released on 21 October 2010. It was the major architecture refactoring: the transition to an OSGi foundation. Since then Roo ensures Roo's add-on infrastructure based on a modular, proven, remote dependency-resolvable classpath management model. Modern IDEs such as Eclipse are also built on OSGi, so this approach for tooling modularity and extensibility is well established.

On 24 April 2014 DISID Corporation got the leadership of the Spring Roo project and it continued the goal of providing a code-generation style of RAD tools, focused on helping developers get Java projects done on time.

Spring Roo 1.3.0 was the first version released by DISID Corporation as project lead, released on 20

November 2014, it included complete support for JDK 8 and it was the first time that Spring Roo jar files had been published to Maven Central!

At the time of writing this document, DISID Spring Roo team was busily working towards the 2.0 release. Spring Roo 2.0 will:

- Create applications based on the extensive set of Spring technologies: Spring Boot, Spring Data, Spring MVC, Spring Security, etc.
- Generate applications based on N-layered architecture pattern: View, Controller, Entity Layer, Service Layer and Repository Layer.
- Generate a fully functional responsive UI based on jQuery and Bootstrap.
- Improve extensibility and increase the collaboration of the Spring Roo project:
  - The "Roo Marketplace": the alternative to Roobot, easier to maintain and available for everyone, a place to find and keep track on third party addons and Roo Addon Suites.
  - A "Roo Addon Suite" is a great way to package and distribute a set of add-ons together.

F.2. Mission Statement

Spring Roo's mission is to "*fundamentally and sustainably improve Java developer productivity without compromising engineering integrity or flexibility*".

Here's exactly what we mean by this:

- "*fundamentally*": We believe a fundamental improvement in developer productivity is attainable. Tools, methodologies and frameworks that offer incidental improvement are nowhere near enough.
- "*and sustainably improve*": A one-off improvement in productivity isn't enough. The productivity improvement needs to sustain beyond the initial jump-start, and continue unabated over a multi-year period. Productivity must remain high even in the face of radically changing requirements, evolving project team membership, and new platform versions
- "*Java developer productivity*": Our focus is unashamedly on developers who work with the most popular programming language in the world, Java. We don't expect Java developers to learn new programming languages and frameworks simply to enjoy a productivity gain. We want to harness their existing Java knowledge, skills and experience, rather than expect them to unlearn what they already know. The conceptual weight must be attainable and reasonable. We always favour evolution over revolution, and provide a solution that is as fun, flexible and intuitive as possible.
- "*without compromising*": Other tools, methodologies and frameworks claim to create solutions that provide these benefits. However, they impose a serious cost in critical areas. We refuse to make this compromise.

- "*engineering integrity*": We embrace OO and language features the way Java language designers intended, greatly simplifying understanding, refactoring, testing and debugging. We don't force projects with significant performance requirements to choose between developer productivity or deployment cost. We move processing to Generation IV web clients where possible, embrace database capabilities, and offer an optimal approach to runtime considerations.

- "*or flexibility*": Projects are similar, but not identical. Developers need the flexibility to use a different technology, pattern or framework when required. While we don't lock developers into particular approaches, we certainly provide an optimal experience when following our recommendations. We ensure that our technology is interface agnostic, gracefully supporting both mainstream IDEs plus the command line. Of course, we support any reasonable deployment scenario, and particularly the emerging class of Generation IV web clients.

We believe that Spring Roo today represents a successful embodiment of this mission statement. While we still have work to do in identified feature areas such as Generation IV web clients, these are easily-achieved future directions upon the existing Roo foundation.

Appendix G: Roo Resources

As an open source project, Spring Roo offers a large number of resources to assist the community learn, interact with one another and become more involved in the project. Below you'll find a short summary of the official project resources.

G.1. Spring Roo Project Home Page

The definitive source of information about Spring Roo is the Spring Roo Home at http://spring.io.

That site provides a brief summary of Roo's main features and links to most of the other project resources. The project home page serves as a hub of information and is the best place to find up-to-date announcements about the project as well as links to articles, blogs and new documentation.

Please use this URI if you are referring other people to the Spring Roo project, as it is the main landing point for the project.

G.2. Downloads and Maven Repositories

You can always access the latest Spring Roo release ZIP by visiting Downloads section at Spring Roo Home Page.

We publish all Roo modules to Maven Central, the default repository from which Maven will download the Spring Roo artifacts automatically.

G.3. Community Forum

Because Roo is an official top-level Spring project, of course you'll find there is a dedicated "Spring Roo" tag at Stack Overflow for all your questions, comments and experiences.

If you have any question about Spring Roo project and its functionalities, you can check and ask a questions at Spring Roo tagged questions at Stack Overflow. We monitor stackoverflow.com for questions tagged with spring-roo.

http://forum.springsource.org is now a read-only archive. All commenting, posting, registration services have been turned off.

The Roo project does not have a "mailing list" or "newsgroup" as you might be familiar with from other open source projects, although commercial support options are available.

Extensive search facilities are provided on the community forums, and the Roo developers routinely answer user questions. One excellent way of contributing to the Roo project is to simply keep an eye on the forum messages and help other people. Even recommendations along the lines of, "I don't know how to do what you're trying to do, but we usually tackle the problem this way instead...." are very

helpful to other community members.

When you ask a question on the forum, it's highly recommended you include a small Roo sample script that can be used to reproduce your problem. If that's infeasible, using Roo's "backup" command is another alternative and you can attach the resulting ZIP file to your post. Other tips include always specifying the version of Roo that you're running (as can be obtained from the "version" command), and if you're having trouble with IDE integration, the exact version of the IDE you are using (and, if an Eclipse-based IDE, the version of AspectJ Development Tools in use). Another good source of advice on how to ask questions on the forum can be found in Eric Raymond's often-cited essay, "How to Ask Smart Questions".

If you believe you have found a bug or are experiencing an issue, it is recommended you first log a message on the forum. This allows other experienced users to comment on whether it appears there is a problem with Roo or perhaps just needs to be used a different way. Someone will usually offer a solution or recommend you log a bug report (usually by saying "please log this in Jira"). When you do log a bug report, please ensure you link to the fully-qualified URI to the forum post. That way the developer who attempts to solve your bug will have background information. Please also post the issue tracking link back in thread you started on the forum, as it will help other people cross-reference the two systems.

G.4. Twitter

Roo Hash Code (please include in your tweets, and also follow for low-volume announcements): #SpringRoo

If you use Twitter, you're encouraged to follow @SpringRoo. Also please use @SpringRoo in your tweets so everyone can easily see them.

The Roo team also uses and monitors tweets that include #SpringRoo, so if you're tweeting about Roo, please remember to include #SpringRoo somewhere in the tweet. If you like Roo or have found it helpful on a project, please tweet about it and help spread the word!

Follow the core Roo development team for interesting Roo news and progress (higher volume than just following @SpringRoo, but only a few Tweets per week): @disid_corp, @juanCaFX, @enrique_ruiz_.

Many people who use Roo also use Twitter, including the core Roo development team. If you're a Twitter user, you're welcome to follow the Roo development team (using the Twitter IDs above) to receive up-to-the-minute Tweets on Roo activities, usage and events.

We do request that you use the Community Forums if you have a question or issue with Roo, as 140 characters doesn't allow us to provide in-depth technical support or provide a growing archive of historical answers that people can search against.

G.5. Issue Tracking

Web: https://jira.spring.io/browse/ROO/

Spring projects use Atlassian Jira for tracking bugs, improvements, feature requests and tasks. Roo uses a public Jira instance you're welcome to use in order to log issues, watch existing issues, vote for existing issues and review the changes made between particular versions.

As discussed in the Community Forums section, we ask that you refrain from logging bug reports until you've first discussed them on the forum. This allows others to comment on whether a bug actually exists. When logging an issue in Jira, there is a field explicitly provided so you can link the forum discussion to the Jira issue.

Please note that every commit into the Roo source repository will be prefixed with a particular Jira issue number. All Jira issue numbers for the Roo project commence with "ROO-", providing you an easy way to determine the rationale of any change.

Because open source projects receive numerous enhancement requests, we generally prioritise enhancements that have patches included, are quick to complete or those which have received a large number of votes. You can vote for a particular issue by logging into Jira (it's fast, easy and free to create an account) and click the "vote" link against any issue. Similarly you can monitor the progress on any issue you're interested in by clicking "watch".

Enhancement requests are easier to complete (and therefore more probable to be actioned) if they represent fine-grained units of work that include as much detail as possible. Enhancement requests should describe a specific use case or user story that is trying to be achieved. It is usually helpful to provide a Roo sample script that can be used to explain the issue. You should also consider whether a particular enhancement is likely to appeal to most Roo users, and if not, whether perhaps writing it as an add-on would be a good alternative.

G.6. Source Repository

Read repository: https://github.com/spring-projects/spring-roo.git

The Git source control system is currently used by Roo for mainline development.

Historical releases of Roo can be accessed by browsing the tags branches within our Git repository. The mainline development of Roo occurs on the "master" branch.

"gh-pages" branch is used to build and publish Spring Roo's project page site based on Jekyll and GitHub Pages.

To detailed information about how to check out and build Roo from Subversion, please refer to the Development Processes chapter.

G.7. Commercial Products and Services

Web: http://www.disid.com/en/

DISID Corporation employs the Roo development team and offers a wide range of products and

professional services around Roo and the technologies which Roo enables. Available professional services include software factory, geographic information systems, web application development, mobile application development, training, consulting and mentoring. Please visit the above URI to learn more about DISID products and services.

Web: http://spring.io/

Pivotal Software offers a wide range of products and professional services around Roo and the technologies which Roo enables. Available professional services include training, consulting, design reviews and mentoring, with products including service level agreement (SLA) backed support subscriptions, certified builds, indemnification and integration with various commercial products. Please visit the above URI to learn more about SpringSource products and services and how these can add value to your build-run-manage application lifecycle.

G.8. Other

Please let us know if you believe it would be helpful to list any other resources in this documentation.