THE EXPERT'S VOICE® IN WEB DEVELOPMENT

The Definitive Guide to Grails 2

Jeff Scott Brown and Graeme Rocher

Apress®

The Definitive Guide to Grails 2

Jeff Scott Brown
Graeme Rocher

Apress®

The Definitive Guide to Grails 2

ISBN 978-1-4302-4377-9
ISBN 978-1-4302-4378-6 (eBook)

President and Publisher: Paul Manning
Lead Editor: Douglas Pundick
Technical Reviewer: Graeme Rocher
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Louise Corrigan, Morgan Ertel, Jonathan Gennick, Jonathan Hassell, Robert Hutchinson, Michelle Lowman, James Markham, Matthew Moodie, Jeff Olson, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Gwenan Spearing, Matt Wade, Tom Welsh
Coordinating Editor: Katie Sullivan
Copy Editor: Thomas McCarthy
Compositor: Bytheway Publishing Services
Indexer: SPi Global
Artist: SPi Global
Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `www.springeronline.com`.

For information on translations, please e-mail `rights@apress.com`, or visit `www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `www.apress.com/bulk-sales`.

Any source code or other supplementary materials referenced by the author in this text is available to readers at `www.apress.com`. For detailed information about how to locate your book's source code, go to `www.apress.com/source-code`.

To Mom.
Thanks for everything.

—Jeff Scott Brown

To Birjinia.
Thanks for being you. Maite zaitut.

—Graeme Rocher

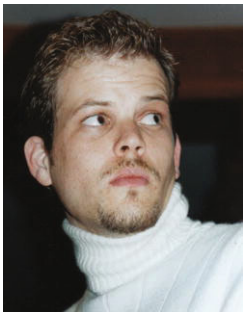
Contents at a Glance

Contents

About the Author

Jeff Scott Brown is an engineer at SpringSource, where he works as a member of the Groovy and Grails development team. Jeff, a technologist for nearly 20 years, has been a member of the Grails team since the framework's very early days. Earlier he was part of G2One, the Groovy/Grails company that eventually became part of SpringSource.

About the Technical Reviewer



Graeme Rocher, a software engineer, a consultant, and an expert in dynamic language, serves as head of Grails Development at SpringSource (www.springsource.com). Graeme is the project lead of the open source Grails web application framework (http://grails.org) and a coauthor of *The Definitive Guide to Grails* (Apress). With Jeff Scott Brown, he is also an author of the present book.

Acknowledgments

First of all, I am grateful to my lovely wife, Betsy, and our boys, Jake and Zack, for all of their support. Without them, none of what I get to do would be possible. Thank you!

To Graeme I have to say a giant thank-you as well. He and I have worked together on the Grails technology for quite a few years, and that experience has been invaluable. I hope we continue enjoying accomplishments together for a very long time.

Thanks, too, to the whole Groovy and Grails team at SpringSource. I have never worked with a smarter group of people or a group that made work seem so much like pleasure.

Thanks as well to the whole Apress team for their support in completing this project. I appreciate their patience and their willingness to help me get this thing done. In particular, thanks to Katie Sullivan, Douglas Pundick, and Steve Anglin for seeing this project through to the end.

Last but not least, I have to extend a big thank-you to Damien Vitrac for contributing some fantastic CSS work to the sample application for this book. The thing looks so much nicer because of his contributions. Well done!

—Jeff Scott Brown

Writing a book is no small task. It requires hours of dedication every day—valuable time stripped away from loved ones. For this alone I thank my wife, Birjinia, whose patience and support drive me to achieve more. Also, thanks to my kids, Alex and Lexeia, who showed remarkable restraint when tempted to wrestle me away from the computer. You guys rock.

To the Grails team at SpringSource, you are a really special group. It continues to be a privilege to work with you all. I count myself extremely lucky to work in the Open Source sector, where cutting-edge innovation and technology leadership are daily occurrences. There is a very special kind of enjoyment that comes from working with such a talented team of innovators.

Thanks to the team at Apress for getting the book done. It is not easy managing all the moving pieces that go into the making of a great technical book. Kudos.

—Graeme Rocher

CHAPTER 1

The Essence of Grails

Simplicity is the ultimate sophistication.

—Leonardo da Vinci

To understand Grails, you first need to understand its goal: to dramatically simplify enterprise Java web development. To take web development to the next level of abstraction. To tap into what has been accessible to developers on other platforms for years. To have all this while still retaining the flexibility to drop down into the underlying technologies and utilize their richness and maturity. Simply put, we Java developers want to "have our cake and eat it, too."

Have you faced the pain of dealing with multiple crippling XML configuration files and an agonizing build system where testing a single change takes minutes instead of seconds? Grails brings back the fun of development on the Java platform, removing barriers and exposing users to APIs that enable them to focus purely on the business problem at hand. No configuration, zero overhead, immediate turnaround.

You might be wondering how you can achieve this remarkable feat. Grails embraces concepts such as Convention over Configuration (CoC), Don't Repeat Yourself (DRY), and sensible defaults that are enabled through the terse Groovy language and an array of domain-specific languages (DSLs) that make your life easier.

As a budding Grails developer, you might think you're cheating somehow, that you should be experiencing more pain. After all, you can't squash a two-hour gym workout into twenty minutes, can you? There must be payback somewhere, maybe in extra pounds?

As a developer you have the assurance that you are standing on the shoulders of giants with the technologies that underpin Grails: Spring, Hibernate, and of course, the Java platform. Grails takes the best of such dynamic language frameworks as Ruby on Rails, Django, and TurboGears and brings them to a Java Virtual Machine (JVM) near you.

This chapter is going to introduce the framework at the highest level and provide some essentials for getting started. All of the concepts introduced here will be explained in detail later in the book.

Simplicity and Power

A factor that clearly sets Grails apart from its competitors is evident in the design choices made during its development. By not reinventing the wheel, and by leveraging tried and trusted frameworks such as Spring and Hibernate, Grails can deliver features that make your life easier without sacrificing robustness.

Grails is powered by some of the most popular open source technologies in their respective categories:

- *Hibernate*: The de facto standard for object-relational mapping (ORM) in the Java world.
- *Spring*: The hugely popular open source Inversion of Control (IoC) container and wrapper framework for Java.
- *SiteMesh*: A robust and stable layout-rendering framework.
- *Tomcat*: A proven, embeddable servlet container.
- *H2*: A pure Java Relational Database Management System (RDBMS) implementation.

The concepts of ORM and IoC might seem a little alien to some readers. ORM simply serves as a way to map objects from the object-oriented world onto tables in a relational database. ORM provides an additional abstraction above SQL, allowing developers to think about their domain model instead of getting wrapped up in reams of SQL.

IoC provides a way of "wiring" together objects so that their dependencies are available at runtime. As an example, an object that performs persistence might require access to a data source. IoC relieves the developer of the responsibility of obtaining a reference to the data source. But don't get too wrapped up in these concepts for the moment, as their usage will become clear later in the book.

You benefit from Grails because it wraps these frameworks by introducing another layer of abstraction via the Groovy language. You, as a developer, will not know that you are building a Spring and Hibernate application. Certainly, you won't need to touch a single line of Hibernate or Spring XML, but it is there at your fingertips if you need it. Figure 1-1 illustrates how Grails relates to these frameworks and the enterprise Java stack.

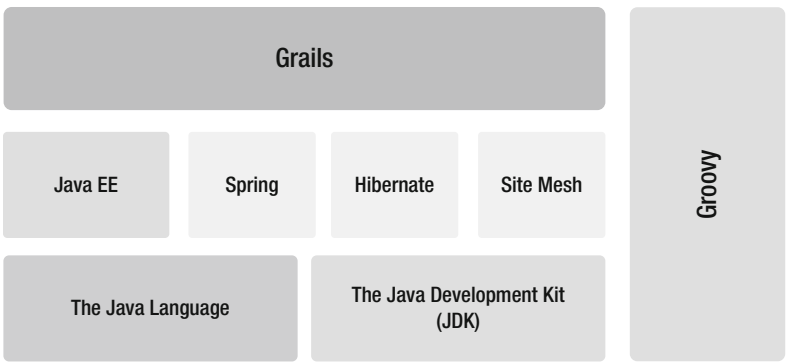


Figure 1-1. *The Grails stack*

Grails, the Platform

When approaching Grails, you might suddenly experience a deep inhalation of breath followed by an outcry of "not another web framework!?" That's understandable, given the dozens of web frameworks that exist for Java. But Grails is different and in a good way. Grails is a full-stack environment, not just a web framework. It is a *platform* with ambitious aims to handle everything from the view layer down to your persistence concerns.

In addition, through its plug-ins system (covered in Chapter 12), Grails aims to provide solutions to an extended set of problems that might not be covered out of the box. With Grails you can accomplish searching, job scheduling, enterprise messaging and remoting, and more.

The sheer breadth of Grails's coverage might conjure up unknown horrors and nightmarish thoughts of configuration, configuration, configuration. However, even in its plug-ins, Grails embraces Convention over Configuration and sensible defaults to minimize the work required to get up and running.

We encourage you to think of Grails as not just another web framework but as the *platform* upon which to build your next web 2.0 phenomenon.

Living in the Java Ecosystem

As well as leveraging Java frameworks that you know and love, Grails gives you a platform that allows you to take full advantage of Java and the JVM—thanks to Groovy. No other dynamic language on the JVM integrates with Java like Groovy. Groovy is designed to work seamlessly with Java at every level. Starting with syntax, the similarities continue as follows:

- The Groovy grammar is derived from the Java 5 grammar, making most valid Java code also valid Groovy code.
- Groovy shares the same underlying APIs as Java, so your trusty javadocs are still valid!
- Groovy objects are Java objects. This has powerful implications that might not be immediately apparent. For example, a Groovy object can implement `Java.io.Serializable` and be sent over Remote Method Invocation (RMI) or clustered using session-replication tools.
- Through Groovy's joint compiler you can have circular references between Groovy and Java without running into compilation issues.
- With Groovy you can easily use the same profiling tools, the same monitoring tools, and all existing and future Java technologies.

Groovy's ability to integrate seamlessly with Java, along with its Java-like syntax, is the number-one reason why its conception generated so much hype. Here was a language with capabilities similar to those of languages such as Ruby and Smalltalk running directly in the JVM. The potential is obvious, and the ability to intermingle Java code with dynamic Groovy code is huge. In addition, Groovy allows mixing of static types and dynamic types, combining the safety of static typing with the power and flexibility to use dynamic typing where necessary.

This level of Java integration is what drives Groovy's continued popularity, particularly in the world of web applications. Across different programming platforms, varying idioms essentially express the same concept. In the Java world there are servlets, filters, tag libraries, and JavaServer Pages (JSP). Moving to a new platform requires relearning all of these concepts and their equivalent APIs or idioms—easy for some, a challenge for others. Not that learning new things is bad, but a cost is attached to knowledge gain in the real world, a cost that can present a major stumbling block in the adoption of any new technology that deviates from the standards or conventions defined within the Java platform and the enterprise.

In addition, Java has standards for deployment, management, security, naming, and more. The goal of Grails is to create a platform with the essence of frameworks like Rails or Django or CakePHP, but one that embraces the mature environment of Java Enterprise Edition (Java EE) and its associated APIs.

Grails is, however, a technology that speaks for itself: the moment you experience using it, a little light bulb will go on inside your head. So without delay, let's get moving with the example application that will flow throughout the course of this book.

The gTunes example will guide you through the development of a music store similar to those provided by Apple, Amazon, and Napster. An application of this nature opens up a wide variety of interesting possibilities, from e-commerce to RESTful APIs and RSS or Atom feeds. We hope it will provide a broad understanding of Grails and its feature set.

Installing and Configuring Grails

Installing Grails is almost as simple as using it, but there is at least one prerequisite to take into account. Grails requires a valid installation of the Java SDK 1.6 or above, which, of course, can be obtained from Oracle: http://www.oracle.com/technetwork/java/javase/.

After installing the Java SDK, set the `JAVA_HOME` environment variable to the location where it is installed and add the `JAVA_HOME/bin` directory to the PATH variables.

■ **Note** If you are working on Mac OS X, you already have Java installed! However, you still need to set `JAVA_HOME` in your ~/.profile file.

To test your installation, open up a command prompt and type java –version:

```
$java -version
```

You should see output similar to Listing 1-1.

Listing 1-1. *Running the Java Executable*

```
java version "1.6.0_29"
Java(TM) SE Runtime Environment (build 1.6.0_29-b11-402-11D50b)
Java HotSpot(TM) 64-Bit Server VM (build 20.4-b02-402, mixed mode)
```

As is typical with many other Java frameworks, including Apache Tomcat and Apache Ant, the installation process involves following a few simple steps. Download and unzip Grails from `http://grails.org`, create a `GRAILS_HOME` variable that points to the location where you installed Grails, and add the `GRAILS_HOME/bin` directory to your `PATH` variable.

To validate your installation, open a command window and type the command `grails -version`:

```
$ grails -version
```

If you have successfully installed Grails, the command will output the usage help shown in Listing 1-2.

Listing 1-2. *Running the Grails Executable*

```
Grails version: 2.1.0
```

Typing `grails help` will display more usage information, including a list of available commands. If more information about a particular command is needed, you can append the command name to the help command. For example, if you want to know more about the `create-app` command, simply type `grails help create-app`:

```
$ grails help create-app
```

Listing 1-3 provides an example of the typical output.

Listing 1-3. *Getting Help on a Command*

```
    grails create-app -- Creates a Grails application for the given name

Usage (optionals in square brackets):

    create-app [--inplace] [NAME]

where
    --inplace  = Creates the project in the current directory rather than
                 creating a new directory.
    NAME       = The name of the project. If not provided, this command will
                 ask you for the name.
```

The Grails command-line interface is built on another Groovy-based project called Gant (`http://gant.codehaus.org/`), which wraps the ever-popular Apache Ant (`http://ant.apache.org/`) build system. Gant allows seamless mixing of Ant targets and Groovy code.

We'll discuss the Grails command line further in Chapter 12.

Creating Your First Application

In this section you're going to create your first Grails application, which will include a simple controller. Here are the steps you'll take to achieve this:

1. Run the command `grails create-app gTunes` to create the application (with "gTunes" being the application's name).
2. Navigate into the `gTunes` directory by issuing the command `cd gtunes`.
3. Create a storefront controller with the command `grails create-controller store`.
4. Write some code to display a welcome message to the user.
5. Test your code and run the tests with `grails test-app`.
6. Run the application with `grails run-app`.

Step 1: Creating the Application

Sound easy? It is, and your first port of call is the `create-app` command; you managed to extract some help with it in the previous section. To run the command, simply type `grails create-app` and hit Enter in the command window:

```
$ grails create-app
```

Grails will automatically prompt you for a project name, as presented in Listing 1-4. When this happens, type `gTunes` and hit Enter. As an alternative, use the command `grails create-app gTunes`, in which case Grails takes the appropriate action automatically.

Listing 1-4. *Creating an Application with the create-app Command*

```
Environment set to development . . .
Application name not specified. Please enter: gTunes
```

Upon completion, the command will have created the gTunes Grails application and the necessary directory structure. The next step is to navigate to the newly created application in the command window using the shell command:

```
cd gTunes
```

At this point you have a clean slate—a newly created Grails application—with the default settings in place. A screenshot of the structure of a Grails application appears in Figure 1-2.

We will delve deeper into the structure of a Grails application and the roles of the various files and directories as we progress through the book. Notice, however, how Grails contains directories for controllers, domain objects (models), and views.

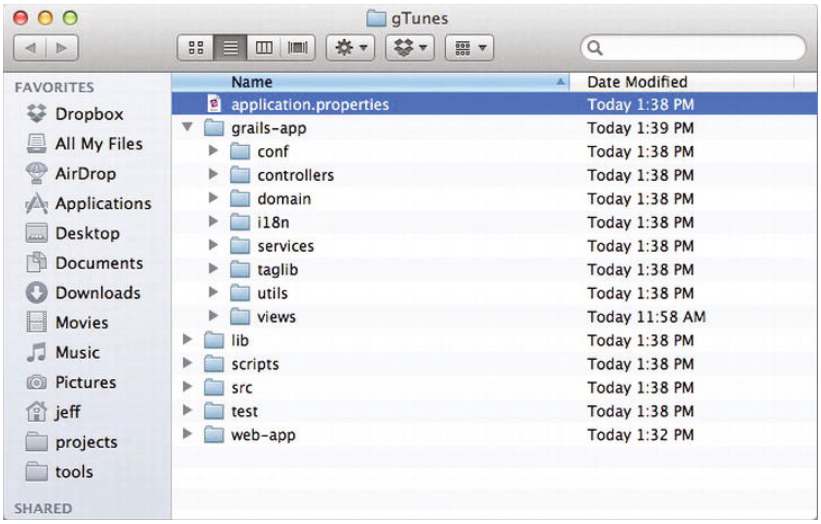


Figure 1-2. *The gTunes application structure*

Step 2: Creating a Controller

Grails is an MVC[1] framework, which means it has models, views, and controllers to separate concerns cleanly. Controllers, which are central to a Grails application, can easily marshal requests, deliver responses, and delegate to views. Because the gTunes application centers on the concept of a music store, we'll show how to create a "store" controller.

To help along the way, Grails features an array of helper commands for creating classes that "fit" into the various slots in a Grails application. For example, for controllers there is the `create-controller` command, which will do nicely. But using these commands is not mandatory. As you grow more familiar with the different concepts in Grails, you can just as easily create a controller class using your favorite text editor or integrated development environment (IDE).

1 The Model-View-Controller (MVC) pattern is a common pattern found in many web frameworks designed to separate user interface and business logic. See Wikipedia, "Model-view-controller," http://en.wikipedia.org/wiki/Model-view-controller, 2003.

Nevertheless, let's get going with the `create-controller` command, which, as with `create-app`, takes an argument where you can specify the name of the controller you wish to create. Simply type `grails create-controller store`:

```
$ grails create-controller store
```

Now sit back while Grails does the rest (see Listing 1-5).

Listing 1-5. *Creating a Controller with the create-controller Command*

```
| Created file grails-app/controllers/gtunes/StoreController.groovy
| Created file grails-app/views/store
| Created file test/unit/gtunes/StoreControllerTests.groovy
```

Once the `create-controller` command has finished running, Grails will have created, not one, but two classes for you: a new controller called `StoreController` within the `grails-app/ controllers` directory and an associated test case in the `test/unit` directory. Since a package name was not specified on the command line, Grails defaults to creating artifacts in a package name that matches the application name. Figure 1-3 shows the newly created controller nesting nicely in the appropriate directory.

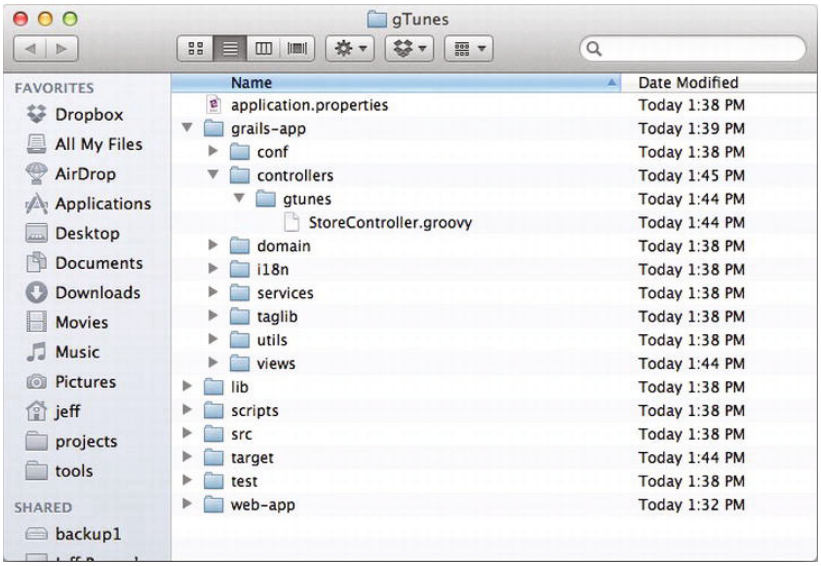


Figure 1-3. *The newly created StoreController*

Due to Groovy's dynamic nature, you should aim for a high level of test coverage[2] in any Grails project (Grails assumes you'll need a test if you're writing a controller). Dynamic languages such as Groovy, Ruby, and Python do not give nearly as much compile-time assistance as a statically typed language such as Java. Some errors that you might expect to be caught at compile time are actually left to runtime, including method resolution. Sadly, the comfort of the compiler often encourages Java developers to forget about testing altogether. Needless to say, the compiler is not a substitute for a good suite of unit tests, and what you lose in compile-time assistance you gain in expressivity.

2 Code coverage is a measure used in software testing. It describes the degree to which the source code of a program has been tested.

Throughout this book we will demonstrate automated-testing techniques that make the most of Grails's testing support.

Step 3: Printing a Message

Let's return to the `StoreController`. By default, Grails will create the controller and give it a single action called `index`. The `index` action is, by convention, the default action in the controller. Listing 1-6 shows the `StoreController` containing the default `index` action.

Listing 1-6. *The Default index Action*

```
package gtunes
class StoreController {
    def index() {}
}
```

The `index` action doesn't seem to be doing much, but by convention its declaration instructs Grails to try to render a view called `grails-app/views/store/index.gsp` automatically. Views are the subject of Chapter 5, so for the sake of simplicity we're going to try something less ambitious instead.

Grails controllers come with a number of implicit methods, which we'll cover in Chapter 4. One of these is `render`, a multipurpose method that, among other things, can render a simple textual response. Listing 1-7 shows how to print a simple response: "Welcome to the gTunes store!"

Listing 1-7. *Printing a Message Using the render Method*

```
package gtunes
class StoreController {
    def index() {
        render 'Welcome to the gTunes store!'
    }
}
```

Step 4: Testing the Code

The preceding code is simple enough, but even the simplest code shouldn't go untested. Open the `StoreControllerTests` test suite that was generated earlier inside the `test/unit` directory. Listing 1-8 shows the contents of the `StoreControllerTests` suite.

Listing 1-8. *The Generated StoreControllerTests Test Suite*

```
package gtunes

import grails.test.mixin.*
import org.junit.*

/**
 * See the API for {@link grails.test.mixin.web.ControllerUnitTestMixin} for usage instructions
 */
@TestFor(StoreController)
class StoreControllerTests {
```

```
    void testSomething() {
       fail "Implement me"
    }
}
```

Grails separates tests into "unit" and "integration" tests. Integration tests bootstrap the whole environment, including the database; hence, they tend to run more slowly. In addition, integration tests are typically designed to test the interaction of a number of classes and therefore require a more complete application before you can run them.

Unit tests, on the other hand, are fast-running tests, but they require extensive use of mocks and stubs. Stubs are classes used in testing that mimic the real behavior of methods by returning arbitrary hard-coded values. Mocks essentially do the same thing but exhibit a bit more intelligence by having "expectations." For example, a mock can specify that it "expects" a given method to be invoked at least once—even ten times if required. As we progress through the book, the difference between unit tests and integration tests will become clearer.

To test the `StoreController` in its current state, assert the value of the response that was sent to the user. A simple way of doing this appears in Listing 1-9.

Listing 1-9. *Testing the StoreController's Index Action*

```
package gtunes

import grails.test.mixin.*
import org.junit.*

/**
 * See the API for {@link grails.test.mixin.web.ControllerUnitTestMixin} for usage instructions
 */
@TestFor(StoreController)
class StoreControllerTests {

    void testSomething() {
        controller.index()
        assert 'Welcome to the gTunes store!' == response.text
    }
}
```

What we're doing here is using the built-in testing capabilities of Grails to evaluate the content of the `response` object. During a test run, Grails magically transforms the regular servlet HttpServletResponse object into a Grails `MockHttpServletResponse`, which has helper properties, such as `text`, that enable you to evaluate what happened as the result of a call to the `render` method.

Nevertheless, don't get too hung up about the ins and outs of using this code just yet. The whole book will be littered with examples; they will gradually ease you into becoming proficient at testing with Grails.

Step 5: Running the Tests

To run the tests and verify that everything works as expected, you can use the `grails test-app` command. The `test-app` command will execute all the tests in the application and output the results to the `test/reports` directory. In addition, you can run only `StoreControllerTests` by issuing the command `grails test-app StoreController`. Listing 1-10 shows some typical output that results when the `grails test-app` command is run.

Listing 1-10. *Running Tests with grails test-app*

```
| Completed 1 unit test, 0 failed in 1107ms
| Tests PASSED - view reports in target/test-reports
```

If you want to review the reports, you'll find XML, HTML, and plain-text reports in the `test/reports` directory. Figure 1-4 shows what the generated HTML reports look like in a browser—they're definitely easier on the eye than the XML equivalent!

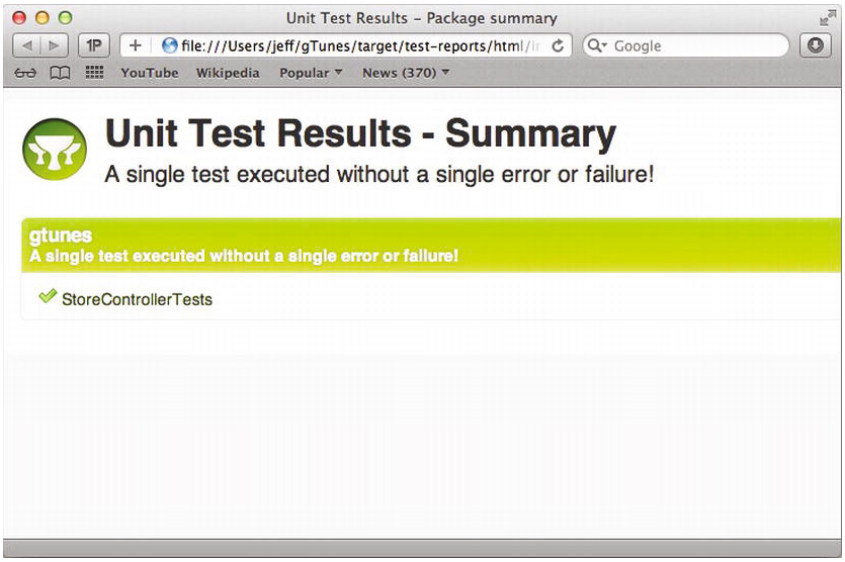


Figure 1-4. *Generated HTML test reports*

Step 6: Running the Application

Now that you've tested your code, the final step is to see it in action. Do this using the `grails run-app` command, which will start up a locally running Grails server on port 8080 by default.

Get Grails going by typing `grails run-app` into the command prompt:

```
$ grails run-app
```

You'll notice that Grails will start up and inform you of a URL you can use to access the Grails instance (see Listing 1-11).

Listing 1-11. *Running an Application with run-app*

```
...
| Server running. Browse to http://localhost:8080/gTunes
```

If you get a bind error, such as the following one, it probably resulted from a port conflict: "Server failed to start: java.net.BindException: Address already in use".

This error typically occurs if you already have another container, such as Apache Tomcat (`http://tomcat.apache.org`), running on port 8080. You can work around this issue by running Grails on a different port by passing the `server.port` argument and specifying an alternative value:

```
grails -Dserver.port=8087 run-app
```

In the preceding case, Grails will start up on port 8087 as expected. Barring any port conflicts, you should have Grails up and running and ready to serve requests at this point. Open your favorite browser and navigate to the URL prompted by the Grails `run-app` command shown in Listing 1-11. You'll be presented with the Grails welcome page that looks something like Figure 1-5.

The welcome screen is (by default) rendered by a Groovy Server Pages (GSP) file located at `web-app/index.gsp`, but you can fully customize the location of this file through URL mappings (discussed in Chapter 6).

As Figure 1-5 shows, the `StoreController` you created earlier is one of those listed as available. Clicking the `StoreController` link results in printing the "Welcome to the gTunes store!" message you implemented earlier (see Figure 1-6).

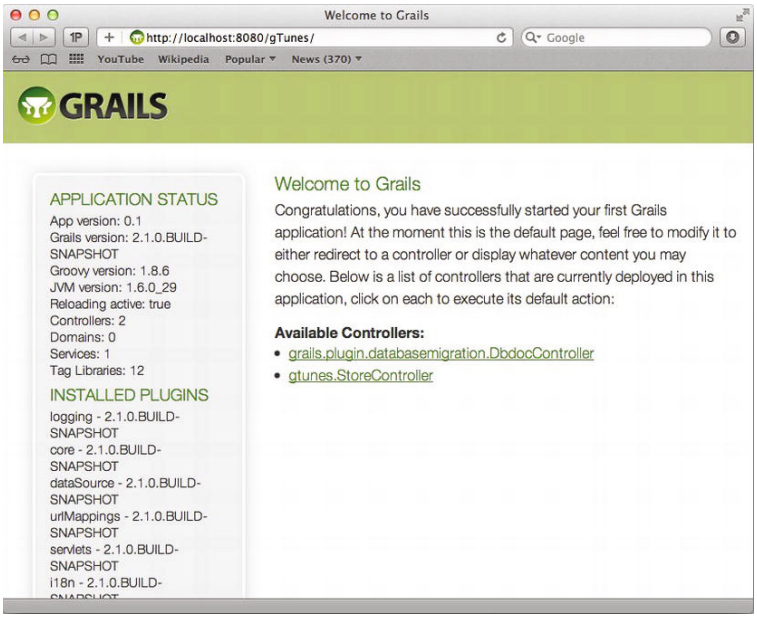


Figure 1-5. *The standard Grails welcome page*

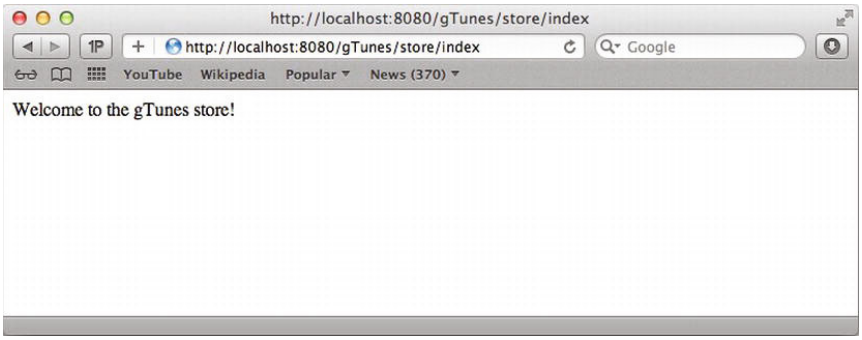


Figure 1-6. *StoreController prints a message.*

Grails Interactive Mode

So far all of the Grails commands that we have seen have been executed by running grails and passing a command name as an argument, for example, `grails create-domain-class`. When a command is executed like this, several things have to happen. The grails command first starts up the JVM, the Groovy runtime environment has to be initialized, and a certain amount of the Grails runtime environment has to be initialized. All of that has to happen before the command can actually be executed and all of that "warming up" takes time. The amount of time will, of course, vary depending on your hardware. Grails "interactive mode" can be a big help here. To start interactive mode, enter the grails command with no arguments. See Listing 1-12.

Listing 1-12. *Starting Interactive Mode*

```
$ grails
grails>
```

As long as you are in the interactive mode, any grails command that could have been executed on the command line may be executed. The syntax is exactly the same as it would be on the command line, except that there is no need to prefix every command with "grails". For example, on the command line you might type something like "grails create-domain-class com.gtunes.Store" but in interactive mode you would type the shorter "create-domain-class com.gtunes.Store" as shown in Listing 1-13.

Listing 1-13. *Creating a Domain Class in Interactive Mode*

```
$ grails
grails> create-domain-class com.gtunes.Store
| Created file grails-app/domain/com/gtunes/Store.groovy
| Created file test/unit/com/gtunes/StoreTests.groovy
grails>
```

Notice that running this command in interactive mode is considerably quicker than running the same command from the command line.

Since pressing the up arrow will cycle through recently executed commands, it's really quick and easy to execute similar commands one after the other. As an example, after executing "create-domain-class com.gtunes.Store", press the up arrow to recall that command and then backspace over "Store" to replace it with "Song" in order to quickly execute "create-domain-class com.gtunes.Song".

Another great productivity boost provided by interactive mode is intuitive tab completion. While in interactive mode, type "cre" followed by pressing Tab, and interactive mode will show all the available commands that start with "cre", as shown in Listing 1-14.

Listing 1-14. *Tab Completion in Interactive Mode*

```
grails> create-

create-controller           create-domain-class         create-filters              create-
hibernate-cfg-xml     create-integration-test      create-plugin               create-
scaffold-controller   create-script               create-service
create-tag-lib              create-unit-test            create-web-xml-config
grails> create-
```

Now that the interactive mode has completed the "cre" command as far as it can—that is, to "create-"—you can type "d" and press Tab again, at which point the interactive mode will complete the command to "create-domain-class", since that is the only available command starting with "create-d". This

same style of autocompletion works for all available commands. Further, some commands that accept arguments also support autocompletion. For example, if you press Tab after "generate-all", since the generate-all command accepts a domain-class name as an argument, the console will show all of the domain classes that are available in the application. This makes it very easy to fill the argument in without typing the full class name. You need to type only enough of the domain-class name to make it unique; the interactive mode can complete the rest.

Interactive mode can help quickly open certain kinds of reports. After generating a domain class or any other artifact, run "test-app unit:" from within interactive mode to run all of the unit tests. The test results will be generated below the project root at target/test-reports/html/index.html. In order to open that report in your default web browser from within interactive mode, use the open command, as shown in Listing 1-15:

Listing 1-15. *Opening Unit Test Report in Interactive Mode*

```
grails> open target/test-reports/html/index.html
grails>
```

Note that tab completion may be used to help complete the path to the HTML file.

It turns out that the open command knows where to find the test report; so a simpler way to open the report is shown in Listing 1-16.

Listing 1-16. *Opening Unit Test Report by Name in Interactive Mode*

```
grails> open test-report
grails>
```

The way to exit the interactive mode is to enter "exit" at the interactive mode prompt. An exception to this occurs if the application is currently running, as it would be after executing "run-app" from the console. In such a case the exit command will exit the application but leave you in interactive mode. At that point you could execute "exit" again to leave interactive mode altogether.

If you are only going to execute a single command, then interactive mode isn't going to be of much use. Interactive mode really benefits the more typical workflow situation where numerous Grails commands are executed over a period of time. You may want to run the tests, view the reports, make some code changes and continue iterating through that loop. You may want to generate several domain classes at once, fill in some of their details, and then generate corresponding controllers and views. Anytime you are going to be executing more than one or two Grails commands during a work session, interactive mode is probably going to be a big help. While doing real development, executing those commands from interactive mode will save you a lot of time.

Interactive mode provides a lot of developer productivity. Getting used to using it will make many development tasks much easier to manage and quicker to execute.

Summary

Success! You have your first Grails application up and running. In this chapter you've taken the first steps toward learning Grails by setting up and configuring your Grails installation. In addition, you've created your first Grails application, along with a basic controller.

Now it is time to see what else Grails does to kick-start your project development. In the chapters that follow, we'll look at some Create, Read, Update, Delete (CRUD) generation facilities, by means of which Grails allows you to flesh out prototype applications in no time.

CHAPTER 2

Getting Started with Grails

In Chapter 1, you got your first introduction to the Grails framework and a feel for the basic command-line interface while creating the basis for the gTunes application. This chapter is going to build on that foundation by showing how you can use the Grails scaffolding feature to quickly build a prototype application that can generate simple CRUD (Create, Read, Update, Delete) interfaces.

Then comes an explanation of some of the basic concepts within the Grails ecosystem, including environments, data sources, and deployment. Get ready—this is an action-packed chapter with loads of information!

What Is Scaffolding?

Scaffolding is a Grails feature that allows you to quickly generate CRUD interfaces for an existing domain. It offers several benefits, the most significant of which is that it serves as a superb learning tool, allowing you to relate how the Grails controller and view layers interact with the domain model that you created.

You should note, however, that Grails is not just a CRUD framework. And scaffolding, although a useful feature in your repertoire, is not the main benefit of Grails. If you're looking for a framework that provides purely CRUD-oriented features, better options are at your disposal.

As with a lot of Grails features, scaffolding is best demonstrated visually, so let's plunge right in and see what can be done.

Creating a Domain

Grails's domain classes serve as the heart of your application and business-model concepts. If you were constructing a bookstore application, for example, you would be thinking about books, authors, and publishers. With gTunes you have albums, artists, songs, and other things in mind.

The most significant attribute that differentiates domain classes from other artifacts within a Grails application is that they are persistent and that Grails automatically maps each domain class onto a physical table in the configured database. (There will be more about how to change the database setup later in the chapter.)

The act of mapping classes onto a relational database layer is also known as object-relational mapping (ORM). The Grails ORM layer, called GORM, is built on the ever-popular Hibernate library (`http://www.hibernate.org`).

Domain classes reside snugly in the `grails-app/domain` directory. You create a domain class by using either the `create-domain-class` command from within interactive mode or your favorite IDE or text editor.