The Definitive Guide to Grails, Second Edition

ISBN-13 (pbk): 978-1-59059-995-2

ISBN-10 (pbk): 1-59059-995-0

ISBN-13 (electronic): 978-1-4302-0871-6

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editors: Steve Anglin, Tom Welsh
Technical Reviewer: Guillaume Laforge
Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Jonathan Hassell, Michelle Lowman, Matthew Moodie, Duncan Parkes, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Kylie Johnston
Copy Editors: Nina Goldschlager, Kim Wimpsett
Associate Production Director: Kari Brooks-Copony
Production Editor: Laura Cheu
Compositor: Pat Christenson
Proofreader: Kim Burton
Indexer: Becky Hornyak
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 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 `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://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 `http://www.apress.com/info/bulksales`.

The source code for this book is available to readers at `http://www.apress.com`.

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 but still retain 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 dynamic language frameworks like Ruby on Rails, Django, and TurboGears and brings them to a Java Virtual Machine (JVM) near you.

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
- *Jetty*: A proven, embeddable servlet container
- *HSQLDB*: 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.

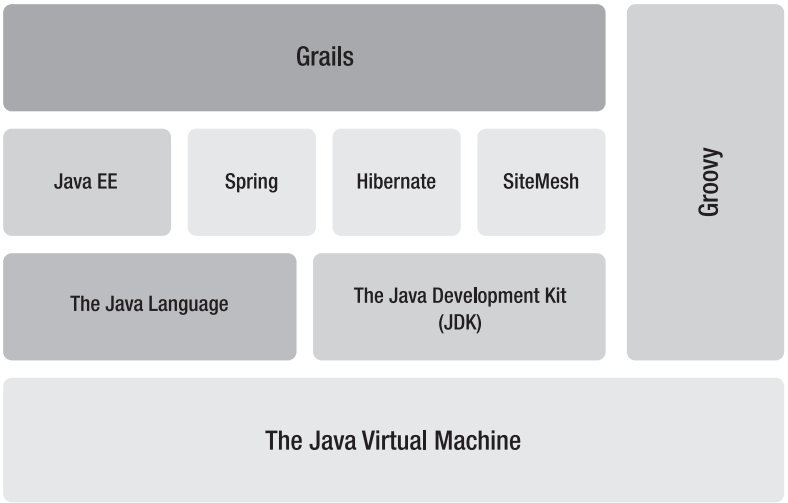

Figure 1-1. *The Grails stack*

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 plugins system (covered in Chapter 13), 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' coverage might conjure up unknown horrors and nightmarish thoughts of configuration, configuration, configuration. However, even in its plugins, 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 the *platform* upon which you plan 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:

- 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 so much hype was generated around its conception. Here we had a language with similar capabilities to 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 you to mix 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 we have 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, which 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, one of these technologies 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. Whereas in this book's first edition we featured a social-bookmarking application modeled on the del.icio.us service, in this edition we'll illustrate an entirely new type of application: gTunes.

Our 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 give you a broad understanding of Grails and its feature set.

Getting Started

Grails' installation is almost as simple as its usage, but you must take into account at least one prerequisite. Grails requires a valid installation of the Java SDK 1.5 or above which, of course, you can obtain from Sun Microsystems at `http://java.sun.com`.

After installing the Java SDK, set the `JAVA_HOME` environment variable to the location where you installed it and add the `JAVA_HOME/bin` directory to your 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 "1.5.0_13"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_13-b05-237)
Java HotSpot(TM) Client VM (build 1.5.0_13-119, mixed mode, sharing)
```

As is typical with many other Java frameworks such as 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**:

```
$ grails
```

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*

```
Welcome to Grails 1.1 - http://grails.org/
Licensed under Apache Standard License 2.0
Grails home is set to: /Developer/grails-1.1

No script name specified. Use 'grails help' for more info or 'grails interactive' to

enter interactive mode
```

As suggested by the output in Listing 1-2, 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*

```
Usage (optionals marked with *):
grails [environment]* create-app

grails create-app -- Creates a Grails project, including the necessary
directory structure and common files
```

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, which you managed to extract some help on 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, you could use the command `grails create-app gTunes`, in which cases Grails takes the appropriate action automatically.

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

```
Running script /Developer/grails-dev/GRAILS_1_1/scripts/CreateApp.groovy
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. You will notice, however, how Grails contains directories for controllers, domain objects (models), and views.

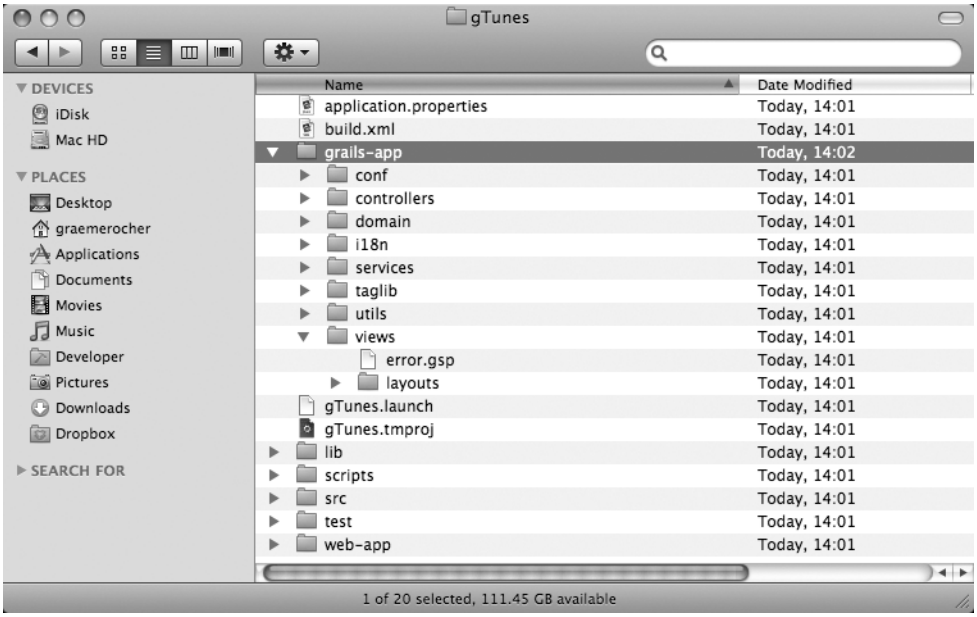

Figure 1-2. *The gTunes application structure*

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 around the concept of a music store, we'll show you how to create a "store" controller.

To help you 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 you have 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).

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).

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`.

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

```
[copy] Copying 1 file to /Developer/grails-dev/gTunes/grails-app/controllers
Created Controller for Store
    [mkdir] Created dir: /Developer/grails-dev/gTunes/grails-app/views/store
     [copy] Copying 1 file to /Developer/grails-dev/gTunes/test/unit
Created ControllerTests for Store
```

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. Figure 1-3 shows the newly created controller nesting nicely in the appropriate directory.

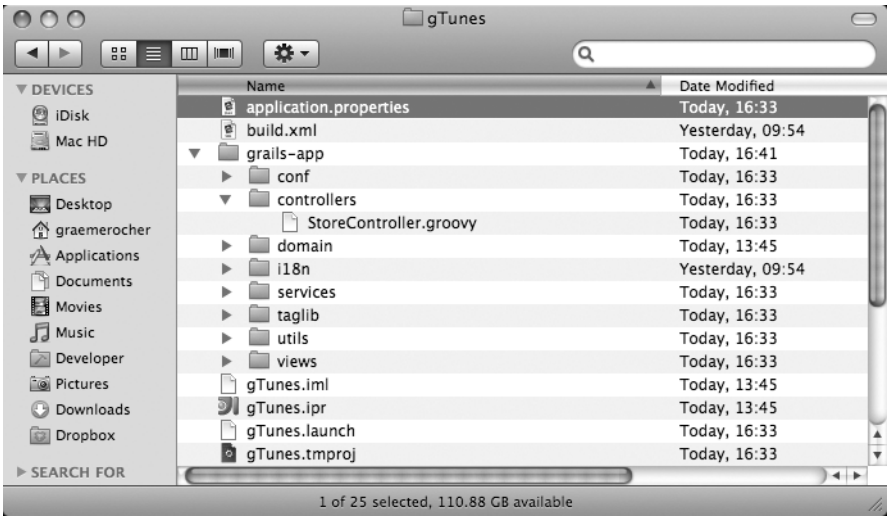

Figure 1-3. *The newly created StoreController*

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 you nearly as much compile-time assistance as statically typed languages 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.

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.

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.

Throughout this book we will be demonstrating automated-testing techniques that make the most of Grails' 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*

```
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*

```
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*

```
class StoreControllerTests extends grails.test.ControllerUnitTestCase {
    void testSomething() {

    }
}
```

Grails separates tests into "unit" and "integration" tests. Integration tests bootstrap the whole environment including the database and hence tend to run more slowly. In addition, integration tests are typically designed to test the interaction among 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 you to make 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, or 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, you can 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*

```
class StoreControllerTests extends grails.test.ControllerUnitTestCase {
    void testRenderHomePage() {
        controller.index()
        assertEquals "Welcome to the gTunes store!",
                                    controller.response.contentAsString
    }
}
```

What we're doing here is using Grails' built-in testing capabilities to evaluate the content of the `response` object. During a test run, Grails magically transforms the regular servlet HttpServletResponse object into a Spring `MockHttpServletResponse`, which has helper properties such as `contentAsString` 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 that 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 you run the `test-app` command.

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

```
-------------------------------------------------------
Running 1 Unit Test...
Running test StoreControllerTests...
                   testRenderHomePage...SUCCESS
Unit Tests Completed in 233ms
-------------------------------------------------------
...
Tests passed. View reports in /Developer/grails-dev/gTunes/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!

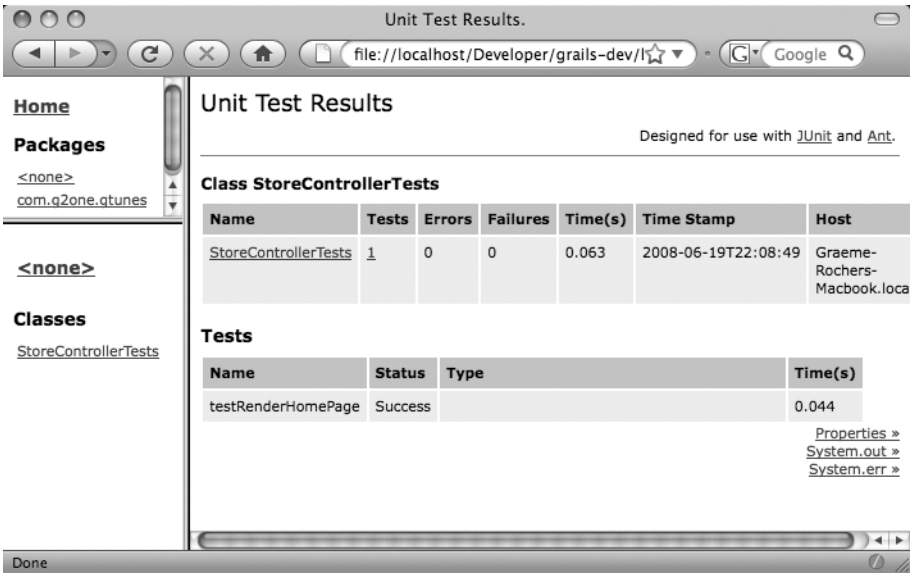

Figure 1-4. *Generated HTML test reports*

Step 6: Running the Application

Now that you've tested your code, your 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. Grails uses the popular and robust Jetty container (`http://www.mortbay.org/`) as the default server, but of course you can deploy Grails onto any servlet container that's version 2.4 or above. (You'll find out more about deployment in Chapter 12.)

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*

```
...
2008-06-19 23:15:46.523:/gTunes:INFO:  Initializing Spring FrameworkServlet 'grails'
2008-06-19 23:15:47.963::INFO:  Started SelectChannelConnector@0.0.0.0:8080
Server running. Browse to http://localhost:8080/gTunes
```

If you get a bind error such as this 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 running on port 8080, such as Apache Tomcat (`http://tomcat.apache.org`). You can work around this issue by running Grails on a different port by passing the `server.port` argument 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 11-1. 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 you can see in Figure 1-5, 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).

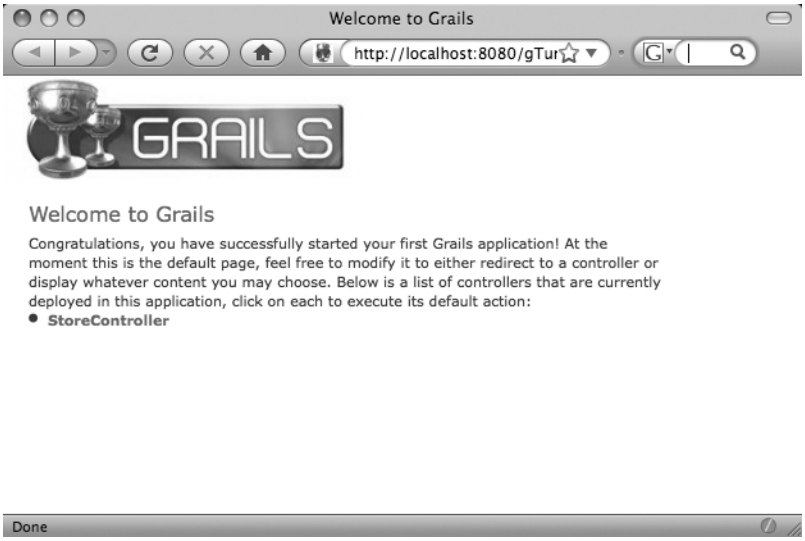

Figure 1-5. *The standard Grails welcome page*

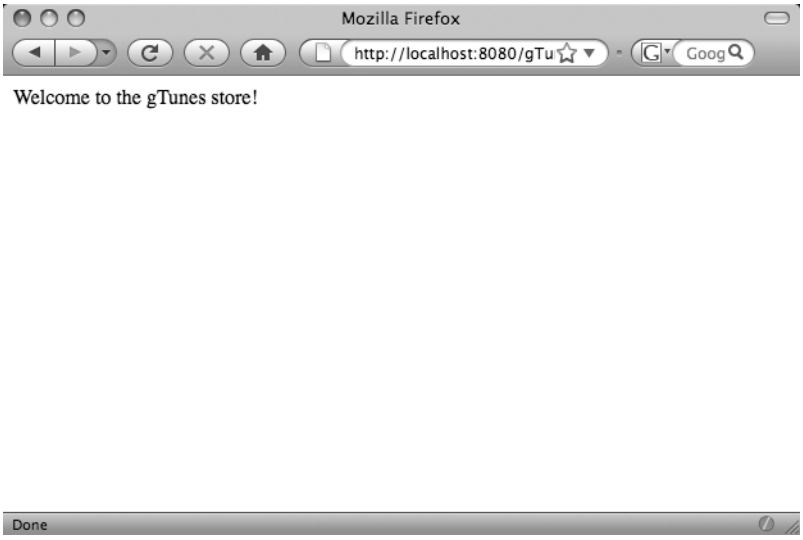

Figure 1-6. *StoreController prints a message.*

Summary

Success! You have your first Grails application up and running. In this chapter you've taken the first steps needed to learn 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 next section, we'll look at some of Grails' Create, Read, Update, Delete (CRUD) generation facilities that allow you to flesh out prototype applications in no time.