SAMPLE CHAPTER

Grails IN ACTION

Glen Smith
Peter Ledbrook

FOREWORD BY Dierk König

MANNING

Grails in Action

by Glen Smith
and Peter Ledbrook

Chapter 1

brief contents

Part 1

Introducing Grails

Great strides have been made in the field of Java-based web application frameworks, but creating a new application with them still seems like a lot of work. Grails' core strength is developing web applications quickly, so we'll jump into writing our first application right away.

In chapter 1, we'll expose you to the core parts of Grails by developing a simple Quote of the Day application from scratch. You'll store and query the database, develop business logic, write tests, and even add some AJAX functionality. By the end of it, you'll have a good feel for all the basic parts of Grails.

In order to develop serious Grails applications, you'll need a firm grasp of Groovy—the underlying dynamic language that makes Grails tick. In chapter 2, we'll take you on a whirlwind tour of core Groovy concepts and introduce you to all the basic syntax.

By the end of part 1, you'll have a real feel for the power of Groovy and Grails and be ready to take on the world. Feel free to do so—Grails encourages experimentation. But you might want to stick around for part 2, where we take you deeper into the core parts of Grails.

1 Grails in a hurry...

This chapter covers

- What is Grails?
- Core Grails philosophy
- Installing Grails
- The key components of a Grails application
- Developing and deploying your first Grails application

"Help, I've lost my Mojo!" That statement is probably the most concise summary of what developers feel when working with one of the plethora of Java web frameworks. So much time editing configuration files, customizing web.xml files, writing injection definitions, tweaking build scripts, modifying page layouts, restarting apps on each change, aaaahhhh! "Where has all the fun gone? Why has everything become so tedious? I just wanted to whip up a quick app to track our customer sign-ups! There's got to be a better way..." We hear you.

Grails is a "next-generation" Java web development framework that draws on best-of-breed web development tooling, techniques, and technologies from existing Java frameworks, and combines them with the power and innovation of dynamic language development. The result is a framework that offers the stability

of technologies you know and love, but shields you from the noisy configuration, design complexity, and boilerplate code that make existing Java web development so tedious. Grails allows you to spend your time implementing features, not editing XML.

But Grails isn't the first player to make such claims. You're probably thinking, "please don't let this be YAJWF (Yet Another Java Web Framework)!" Because if there's one thing that the Java development world is famous for, it's for having an unbelievably large number of web frameworks available. Struts, WebWork, JSF, Spring MVC, Seam, Wicket, Tapestry, Stripes, GWT, and the list goes on and on—all with their own config files, idioms, templating languages, and gotchas. And now we're introducing a new one?

The good news is that this ain't your Grandma's web framework—we're about to take you on a journey to a whole new level of getting stuff done—and getting it done painlessly. We're so excited about Grails because we think it's time that Java web app development was fun again! It's time you were able to sit down for an afternoon and crank out something you'd be happy demoing to your boss, client, or the rest of the internet. Grails is that good.

In this chapter, we're going to take you through developing your first Grails app. Not a toy, either. Something you could deploy and show your friends. An app that's data-driven and Ajax-powered, and that has full CRUD (create, read, update, delete) implementation, a template-driven layout, and even unit tests. In half an hour, with less than 100 lines of code. Seriously.

But before we get our hands dirty writing code, you may need a little more convincing as to why Grails should be on your radar. Before you fire up your IDE, let's quickly review the history to learn why Grails is such a game-changer.

1.1 Why Grails?

Grails is a next-generation Java web development framework that generates great developer productivity gains through the confluence of a dynamic language, a Convention over Configuration philosophy, powerfully pragmatic supporting tools, and an agile perspective drawn from the best emerging web development paradigms.

1.1.1 First there was Rails...

Some have incorrectly labeled Grails a port of Ruby on Rails to the Java platform, but this fails to recognize several points about Grails:

- The amazing innovations that Grails, itself, has brought to the enterprise development sector with its own secret sauces
- The broad range of platforms that have influenced Grails (which include Ruby, Python, PHP, and Java frameworks)
- The many features that Grails brings to the table that aren't presently available in Rails—features drawn from the JVMs long history of use in enterprise settings

Nevertheless, Grails does embrace many of the innovative philosophies that Rails brought to web development. When Ruby on Rails hit the web development landscape (in 2004), and started gaining real industry traction and critical acclaim (during 2006), a whole new set of ideas about web development started to germinate.

None of the ideas were particularly new, but the execution was truly stunning. Things like Convention over Configuration, scaffolding, code templates, and easy database integration made bootstrapping an application lightning fast. The killer demo was when David Heinemeier Hansson (the Rails founder) developed a database-driven blog application from scratch in 15 minutes. Everyone's jaw dropped.

The real power of these ideas was brought to the fore in Rails by using a dynamic language (Ruby) to perform amazing metaclass magic. For those of us in enterprise Java-land, there wasn't a compelling Java equivalent. We were stuck with a statically typed language that didn't give us the same agility to do the metaclass work that made it all work so elegantly.

1.1.2 Why Grails changed the game

Then, in 2006, along came Grails. Taking full advantage of Groovy as the underlying dynamic language, Grails made it possible to create a `Book` object and query it with dynamic methods like `Book.findByTitle("Grails in Action")` or `Book.findAllByDatePublishedGreaterThanAndTitleLike(myDate, "Grails")`, even though none of those methods really existed on the `Book` object.

Even better, you could also access any Java code or libraries you were already using, and the language syntax was similar enough to Java to make the learning curve painless. But best of all, at the end of the day, you had a WAR file to deploy to your existing Java app server—no special infrastructure required, and no management awareness needed.

The icing on the cake was that Grails was built on Spring, Hibernate, and other libraries already popular in enterprise Java—the stuff developers were already building applications on. It was like turbo-charging existing development practices without sacrificing reliability or proven technologies.

Grails' popularity exploded. Finally Java web developers had a way to take all the cool ideas that Rails had brought to the table and apply them to robust enterprise-strength web application development, without leaving any of their existing skills, libraries, or infrastructure behind.

That's probably enough history about how Grails ended up being such a popular Java web framework. But if you (or your manager) need further convincing about why Grails is an outstanding option for your next big web app project, the following subsections discuss seven of the big ideas (shown in figure 1.1) that have driven Grails to such a dominant place in the emerging next-gen Java web frameworks market.

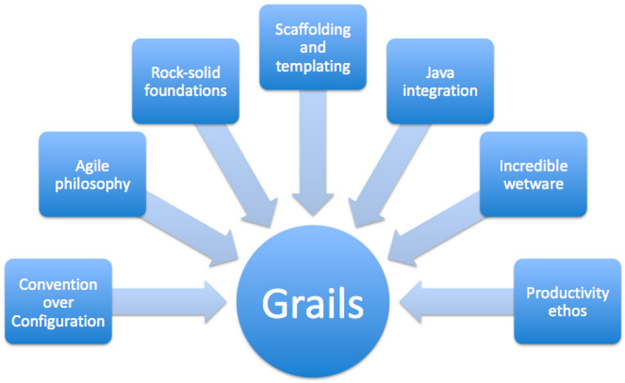

Figure 1.1 The Grails ecosystem is a powerful confluence of people, ideas, and technology.

1.1.3 Big idea #1: Convention over Configuration

One of the things you'll notice about developing with Grails is how few configuration files there are. Grails makes most of its decisions based on sensible defaults drawn from your source code:

- Add a controller class called `Shop` with an action called `order`, and Grails will expose it as a URL of /yourapp/shop/order.
- Place your view files in a directory called /views/shop/order, and Grails will look after linking everything up for you without a single line of configuration.
- Create a new domain class called `Customer`, and Grails will automatically create a table called customer in your database.
- Add some fields to your `Customer` object, and Grails will automatically create the necessary fields in your customer table on the fly (including the right data types based on the validation constraints you place on them). No SQL required.

But as Jason Rudolph is quick to point out, Grails is about Convention *over* Configuration, not Convention *instead of* Configuration. If you need to tweak the defaults, all the power is there for you to do so. Grails makes overriding the defaults easy, and you still won't need any XML. But if you want to use your existing Hibernate configuration XML files in all their complex glory, Grails won't stand in your way.

1.1.4 Big idea #2: agile philosophy

Grails makes a big deal about being an agile web framework, and by the time you finish this chapter, you'll understand why. By making use of a dynamic language (Groovy), Grails makes things that were once a real pain in Java a complete joy. Whether it's processing form posts, implementing tag libraries, or writing test cases, there's a conciseness and expressiveness to the framework that makes these operations both easier and more maintainable at the same time.

The Grails infrastructure adds to the pleasure by keeping you iterating without getting in the way. Imagine starting up a local copy of your application and adding controllers, views, and taglib features while it's running—without having to restart it! Then imagine testing those features, making tweaks, and clicking refresh in your browser to view the updates. It's a joy.

Grails brings a whole new level of agility to Java web application development, and once you've developed your first complete application, which you'll do over the next 30 minutes or so, you'll start to appreciate some of the unique power Grails provides.

1.1.5 Big idea #3: rock-solid foundations

Even though Grails itself is full of innovation and cutting-edge ideas, the core is built on rock-solid proven technologies: Spring and Hibernate. These are the technologies that many existing Java shops are using today, and for good reason: they're reliable and battle tested.

Building on Spring and Hibernate also means that there's very little magic going on under the hood. If you need to tweak things in the configuration (by customizing a Hibernate configuration class) or at runtime (by getting a handle to a Spring `ApplicationContext`), there's no new magic to learn. None of your learning time on Spring and Hibernate has been wasted.

If you're new to Grails and don't have a background in Spring and Hibernate, it doesn't matter. There are few Grails development cases where you need to fall back to that level anyway, but you can feel good knowing it's there if you need it.

This same philosophy of using best-of-breed components has translated to other areas of the Grails ecosystem—particularly third-party plugins. The scheduling plugin is built on Quartz, the search plugin is built on Lucene and Compass, and the layout engine is built on SiteMesh. Wherever you go in the ecosystem, you'll see popular Java libraries wrapped in an easy-to-use instantly productive plugin. Peace of mind plus amazing productivity!

Another important part of the foundation for enterprise developers is having the formal backing of a professional services, training, and support organization. When SpringSource acquired G2One in November 2008, Groovy and Grails inherited the backing of a large company with deep expertise in the whole Groovy and Grails stack. This also introduced a range of support options to the platform useful to those organizations looking for 24/7 Groovy and Grails support backup.

1.1.6 Big idea #4: scaffolding and templating

If you've ever tried bootstrapping a Spring MVC application by hand, you'll know that it isn't pretty. You'll need a directory of JAR files, a bunch of bean definition files, a set of web.xml customizations, a bunch of annotated POJOs, a few Hibernate configuration files, a database-creation script, and then a build system to turn it all into a running application. It's hard work, and you'll probably burn a day in the process.

By contrast, building a running Grails application is a one liner: `grails create-app myapp`, and you can follow it up with `grails run-app` to see it running in your browser. All of the same stuff is happening behind the scenes, but based on conventions and sensible defaults rather than on hand-coding and configuration.

If you need a new controller class, `grails create-controller` will generate a shell for you (along with a shell test case). The same goes for views, services, domain classes, and all of the other artifacts in your application. This template-driven approach bootstraps you into a fantastic level of productivity, where you spend your time solving problems, not writing boilerplate code.

Grails also offers an amazing feature called "scaffolding." Based on the fields in your database model classes, Grails can generate a set of views and controllers on the fly to handle all your basic CRUD operations—creating, reading, updating, and deleting—without a single line of code.

1.1.7 Big idea #5: Java integration

One of the unique aspects of the Groovy and Grails community is that, unlike some of the other JVM languages, we love Java! We appreciate that there are problems and design solutions that are much better implemented in a statically typed language, so we have no problem writing our web form processing classes in Groovy, and our high-performance payroll calculations in Java. It's all about using the right tool for the job.

We're also in love with the Java ecosystem. That means we don't want to leave behind the amazing selection of Java libraries we know and love. Whether that's in-house DTO JARs for the payroll system, or a great new Java library for interfacing with Facebook, moving to Grails means you don't have to leave anything behind—except a lot of verbose XML configuration files. But as we've already said, you can reuse your Hibernate mappings and Spring resource files if you're so inclined!

1.1.8 Big idea #6: incredible wetware

One of the most compelling parts of the Grails ecosystem is the fantastic and helpful user community. The Groovy and Grails mailing list is a hive of activity where both die-hard veterans and new users are equally welcome. The Grails.org site hosts a Grails-powered wiki full of Grails-related information and documentation.

A wealth of third-party community websites have also sprung up around Grails. For example, groovyblogs.org aggregates what's happening in the Groovy and Grails blogosphere and is full of interesting articles. And sites like grailscrowd.com, Facebook, and LinkedIn host Grails social networking options. There's even a Grails podcast (grailspodcast.com) that runs every two weeks to keep you up to date with news, interviews, and discussions in the Grails world.

But one of the coolest parts of the community is the amazing ever-growing list of third-party plugins for Grails. Whether it's a plugin to implement full-text search, Ajax widgets, reporting, instant messaging, or RSS feeds, or to manage log files, profile performance, or integrate with Twitter, there's something for everyone. There are

literally hundreds of time-saving plugins available (and in chapter 8, we'll introduce you to a bunch of the most popular ones).

1.1.9 Big idea #7: productivity ethos

Grails isn't just about building web applications—it's about executing your vision quickly so you can get on with doing other "life stuff" that's more important. For us, productivity is the new black, and developing in Grails is about getting your life back, one feature at a time. When you realize that you can deliver work in one day that used to take you two weeks, you start to feel good about going home early. Working with such a productive framework even makes your hobby time more fun. It means you can complete all those Web 2.0 startup website ideas you've dreamed about, but which ended up as half-written Struts or Spring MVC apps.

Developing your applications quickly and robustly gives you more time to do other, more important stuff: hanging out with your family, walking your dog, learning rock guitar, or getting your veggie patch growing really big zucchinis. Web apps come and go; zucchinis are forever. Grails productivity gives you that sort of sage-like perspective. Through the course of this chapter, we'll give you a taste of the kind of productivity you can expect when moving to Grails.

Most programmers we know are the impatient type, so in this chapter we'll take 30 minutes to develop a data-driven, Ajax-powered, unit-tested, deployable Web 2.0 website. Along the way, you'll get a taste of the core parts of a Grails application: models, views, controllers, taglibs, and services. Buckle up—it's time to hack.

1.2 Getting set up

In order to get Grails up and running, you'll need to walk through the installation process shown in figure 1.2.

First, you'll need to have a JDK installed (version 1.5 or later—run `javac -version` from your command prompt to check which version you have). Most PCs come with Java preinstalled these days, so you may be able to skip this step.

Once you're happy that your JDK is installed, download the latest Grails distro from www.grails.org and unzip it to your favorite installation area.

You'll then need to set the `GRAILS_HOME` environment variable, which points to your Grails installation directory, and add GRAILS_HOME/bin to your path. On Mac OS X and Linux, this is normally done by editing the ~/.profile script to contain lines like these:

```
export GRAILS_HOME=/opt/grails
export PATH=$PATH:$GRAILS_HOME/bin
```

On Windows, you'll need to go into System Properties to define `GRAILS_HOME` and update your `PATH` setting.

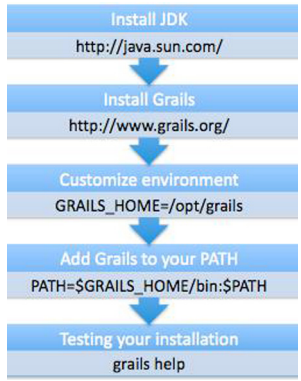

Figure 1.2 The Grails installation process

You can verify that Grails is installed correctly by running `grails help` from the command line. This will give you a handy list of Grails commands, and it'll confirm that everything is running as expected and that your `GRAILS_HOME` is set to a sensible location:

```
grails help
```

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

Looks like everything is in good working order.

When you develop more sophisticated Grails applications, you'll probably want to take advantage of some of the fantastic Grails IDE support out there. There's now Grails plugin support for IntelliJ, NetBeans, and Eclipse—whichever your preferred IDE, there will be a plugin to get you going. We won't be developing too much code in this chapter, so a basic text editor will be all you need. Fire up your favorite editor, and we'll talk about our sample application.

1.3 Our sample program: a Web 2.0 QOTD

If we're going to the trouble of writing a small application, we might as well have some fun. Our example is a quote-of-the-day (QOTD) web application where we'll capture and display famous programming quotes from development rock stars throughout time. We'll let the user add, edit, and cycle through programming quotes, and we'll even add some Ajax sizzle to give it a Web 2.0 feel. We'll want a nice short URL for our application, so let's make "qotd" our application's working title.

NOTE You can download the sample apps for this book, including CSS and associated graphics, from the book's site at manning.com.

It's time to get started world-changing Web 2.0 quotation app, and all Grails projects begin the same way. First, find a directory to work in. Then create the application:

```
grails create-app qotd
cd qotd
```

Well done. You've created your first Grails application. You'll see that Grails created a qotd subdirectory to hold our application files. Change to that directory now, and we'll stay there for the rest of the chapter.

Because we've done all the hard work of building the application, it'd be a shame not to enjoy the fruit of our labor. Let's give it a run:

```
grails run-app
```

Grails ships with a copy of Jetty (an embeddable Java web server—there is talk that a future version will switch to Tomcat), which Grails uses to host your application during the development and testing lifecycle. When you run the `grails run-app` command, Grails will compile and start your web application. When everything is ready to go, you'll see a message like this on the console:

```
Server running. Browse to http://localhost:8080/qotd
```

This means it's time to fire up your favorite browser and take your application for a spin: http://localhost:8080/qotd/. Figure 1.3 below shows our QOTD application up and running in a browser.

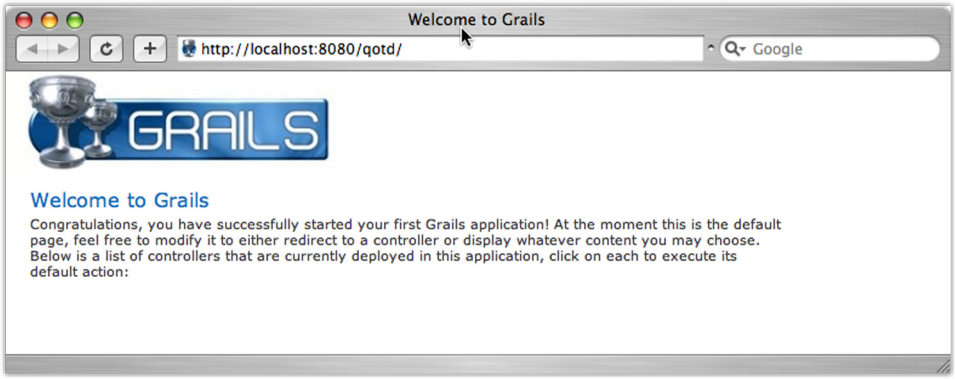

Figure 1.3 Our first app is up and running.

Once you've taken in the home page, you can stop the application by pressing Ctrl-C. Or you can leave it running and issue Grails commands from a separate console window in your operating system.

> **Running on a custom port (not 8080)**
>
> If port 8080 is just not for you (because perhaps you have another process running there, like Tomcat), you can customize the port that the Grails embedded application server runs on using the `-Dserver.port` command-line argument. If you want to run Grails on port 9090, for instance, you could run your application like this:
>
> ```
> grails -Dserver.port=9090 run-app
> ```
>
> If you decide to always run a particular application on a custom port, you can create a custom /grails-app/conf/BuildConfig.groovy file with an entry for `grails.server.port.http=9090` to make your custom port the default. Or make a system-wide change by editing the global $HOME/.grails/settings.groovy file. You'll find out more about these files in chapter 15.

1.3.1 *Writing your first controller*

We have our application built and deployed, but we're a little short on an engaging user experience. Before we go too much further, now's a good time to learn a little about how Grails handles interaction with user—that's via a *controller*.

Controllers are at the heart of every Grails application. They take input from your user's web browser, interact with your business logic and data model, and route the

user to the correct page to display. Without controllers, your web app would be a bunch of static pages.

Like most parts of a Grails application, you can let Grails generate a skeleton controller by using the Grails command line. Let's create a simple controller for handling quotes:

```
grails create-controller quote
```

Grails will create this skeleton controller in /grails-app/controllers/QuoteController.groovy. You'll notice that Grails sorted out the capitalization for you. The basic skeleton is shown in listing 1.1.

Listing 1.1 Our first quote controller

```
class QuoteController {

   def index = { }
}
```

Not so exciting, is it? The index entry in listing 1.1 is a Grails *action*, which we'll return to in a moment. For now, let's add a home action that sends some text back to the browser—it's shown in listing 1.2.

Listing 1.2 Adding some output

```
class QuoteController {

   def index = { }

   def home = {
      render "<h1>Real Programmers do not eat Quiche</h1>"
   }
}
```

Grails provides the `render()` method to send content directly back to the browser. This will become more important when we dip our toes into Ajax waters, but for now let's use it to deliver our "Real Programmers" heading.

How do we invoke our action in a browser? If this were a Java web application, the URL to get to it would be declared in a configuration file, but not in Grails. This is where we need to introduce you to the Convention over Configuration pattern.

Ruby on Rails introduced the idea that tons of XML configuration (or configuration of any sort) can be avoided if the framework makes some opinionated choices for you about how things will fit together. Grails embraces the same philosophy. Because our controller is called `QuoteController`, Grails will expose its actions over the URL /qotd/quote/youraction. The following gives a visual breakdown of how URLs translate to Grails objects.

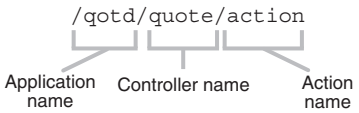

Figure 1.4 Adding our first bit of functionality

In the case of our `hello` action, we'll need to navigate to this URL:

```
http://localhost:8080/qotd/quote/home
```

Figure 1.4 shows our brand new application up and running, without a single line of XML.

If you were wondering about that `index()` routine in listing 1.1, that's the method that's called when the user omits the action name. If we decide that all references to /qotd/quote/ should end up at /qotd/quote/home, we need to tell Grails about that with an index action, like the one in listing 1.3.

Listing 1.3 Handling redirects

```
class QuoteController {

   def index = {
      redirect(action: home)
   }

   def home = {
      render "<h1>Real Programmers do not each quiche!</h1>"
   }
}
```

It's looking pretty good so far, but it's pretty nasty to have that HTML embedded in our source. Now that we've learned a little about controllers, it's time to get acquainted with views.

1.3.2 Writing stuff out: the view

Embedding HTML inside your code is always a bad idea. Not only is it difficult to read and maintain, but your graphic designer will need access to your source code in order to design the pages. The solution is to move your display logic out to a separate file, which is known as the *view*, and Grails makes it simple.

If you've done any work with Java web applications, you'll be familiar with JavaServer Pages (JSP). JSPs render HTML to the user of your web application. Grails applications, conversely, make use of Groovy Server Pages (GSP). The concepts are quite similar.

We've already discussed the Convention over Configuration pattern, and views take advantage of the same stylistic mindset. If we create our view files in the right place, everything will hook up without a single line of configuration.

First, in listing 1.4, we implement our random action. Then we'll worry about the view.

Listing 1.4 A random quote action

```
def random = {
   def staticAuthor = "Anonymous"
   def staticContent = "Real Programmers don't eat much quiche"
   [ author: staticAuthor, content: staticContent]
}
```

What's all that square bracket-ness? That's how the controller action passes information to the view. If you're an old-school servlet programmer, you might think of it as request-scoped data. The `[:]` operator in Groovy creates a `Map`, so we're passing a series of key/value pairs through to our view.

Where does our view fit into this, and where will we put our GSP file so that Grails knows where to find it? We'll use the naming conventions we used for the controller, coupled with the name of our action, and we'll place our GSP in /grails-app/views/quote/random.gsp. If we follow that pattern, there's no configuration required.

Let's create a GSP file and see how we can reference our `Map` data, as shown in listing 1.5.

Listing 1.5 Implementing our first view

```
<html>
<head>
   <title>Random Quote</title>
</head>

<body>

   <q>${content}</q>
   <p>${author}</p>

</body>

</html>
```

The `${content}` and `${author}` format is known as the GSP Expression Language, and if you've ever done any work with JSPs, it will probably be old news to you. If you haven't worked with JSPs before, you can think of those `${}` tags as a way of displaying the contents of a variable. Let's fire up the browser and give it a whirl. Figure 1.5 shows our new markup in action.

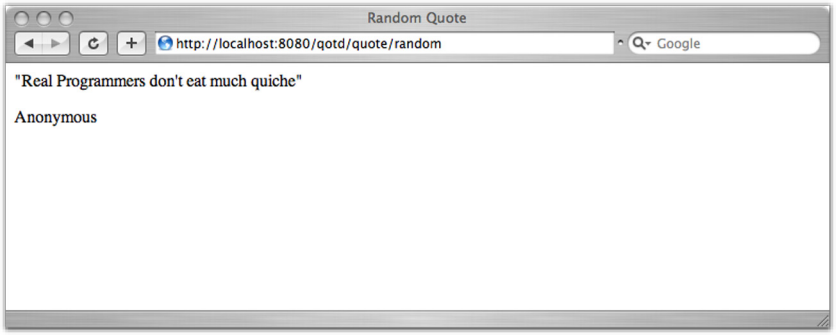

Figure 1.5 Our first view in action

1.3.3 *Adding some style with Grails layouts*

We now have our first piece of backend functionality written. But the output isn't engaging—there are no gradients, no giant text, no rounded corners. Everything looks pretty mid-90s.

You're probably thinking it's time for some CSS action, but let's plan ahead a little. If we mark up random.gsp with CSS, we're going to have to add those links to the header of every page in the app. There's a better way: Grails layouts.

Layouts give you a way of specifying layout templates for certain parts of your application. For example, we might want all of the quote pages (random, by author, by date) to be styled with a common masthead and navigation links; only the body content should change. To do this, let's first mark up our target page with some IDs that we can use for our CSS. This is shown in listing 1.6.

Listing 1.6 Updating the view

```
<html>
<head>
   <title>Random Quote</title>
</head>

<body>

   <div id="quote">
      <q>${content}</q>
      <p>${author}</p>
   </div>

</body>

</html>
```

Now, how can we apply those layout templates (masthead and navigation) we were discussing earlier? Like everything else in Grails, layouts follow a Convention over Configuration style. To have all our `QuoteController` actions share the same layout, we'll create a file called /grails-app/views/layouts/quote.gsp. There are no Grails shortcuts

for layout creation, so we've got to roll this one by hand. Listing 1.7 shows our first attempt at writing a layout.

Listing 1.7 Adding a layout

```
<html>
   <head>
      <title>QOTD &raquo; <g:layoutTitle/></title>
      <link rel="stylesheet" href="
            <g:createLinkTo dir='css' file='snazzy.css' />
      " />
      <g:layoutHead />
   </head>
   <body>
      <div id="header">
         <img src="
            <g:createLinkTo dir='images' file='logo.png'/>
         " alt="logo"/>
      </div>
      <g:layoutBody />
   </body>
</html>
```

1. Merges title from our target page
2. Creates relative link to CSS file
3. Merges head elements from target page
4. Merges body elements from target page

That's a lot of angle brackets—let's break it down. The key thing to remember is that this is a template page, so the contents of our target page (random.gsp) will be merged with this template before we send any content back to the browser. Under the hood, Grails is using SiteMesh, the popular Java layout engine, to do all of that merging for you. The general process for how SiteMesh does the merge is shown in figure 1.6.

In order for our layout template in listing 1.7 to work, it needs a way of accessing elements of the target page (when we merge the title of the target page with the template, for example). That access is achieved through Grails' template taglibs, so it's probably time to introduce you to the notion of taglibs in general.

If you've never seen a tag library (taglib) before, think of them as groups of custom HTML tags that can execute code. In listing 1.7, we took advantage of the `g:createLinkTo`, `g:layoutHead`, and `g:layoutBody` tags. When the client's browser requests the page, Grails replaces all of those tag calls with real HTML, and the contents of the

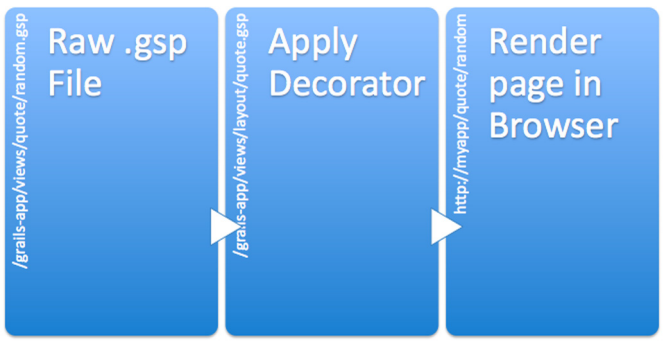

Figure 1.6 SiteMesh decorates a raw GSP file with a standard set of titles and sidebars.

HTML will depend on what the individual tag generates. For instance, that first `createLinkTo` tag ❷ will end up generating a link fragment like /qotd/css/snazzy.css.

In the title block of the page, we include our QOTD title and then follow it with some chevrons (>>) represented by the HTML character code `&raquo;`, and then add the title of the target page itself ❶.

After the rest of the head tags, we use a `layoutHead` call to merge the contents of the `HEAD` section of any target page ❸. This can be important for search engine optimization (SEO) techniques, where individual target pages might contain their own `META` tags to increase their Google-ability.

Finally, we get to the body of the page. We output our common masthead div to get our Web 2.0 gradient and cute icons, and then we call `<g:layoutBody>` to render the `BODY` section of the target page ❹.

Let's refresh our browser to see how we're doing. Figure 1.7 shows our styled page.

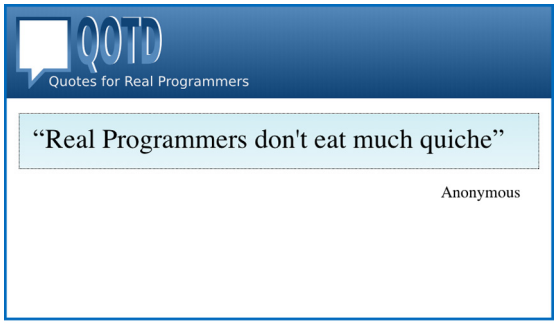

Figure 1.7 QOTD with some funky CSS skinning

Our app is looking good. Notice how we've made no changes to our relatively bland random.gsp file. Keeping view pages free of cosmetic markup reduces your maintenance overhead significantly. And if you need to change your masthead, add some more JavaScript includes, or incorporate a few additional CSS files. You do it all in one place: the template.

Fantastic. We're up and running with a controller, view, and template. But things are still pretty static in the data department. We're probably a little overdue to learn how Grails handles stuff in the database. Once we have that under our belt, we can circle back and implement a real random action.

1.4 Creating the domain model

We've begun our application, and we can deploy it to our testing web container. But let's not overstate our progress—Google isn't about to buy us just yet. Our app lacks a certain pizzazz. It's time to add some interactivity so that our users can add new quotations to the database. To store those quotations, we're going to need to learn how Grails handles the data model.

Grails uses the term "domain class" to describe those objects that can be persisted to the database. In our QOTD app, we're going to need a few domain classes, but let's start with the absolute minimum: a domain class to hold our quotations.

Let's create a `Quote` domain class:

```
grails create-domain-class quote
```

In your Grails application, domain classes always end up under /grails-app/domain. Take a look at the skeleton class Grails has created in /grails-app/domain/Quote.groovy:

```
class Quote {

  static constraints = {
  }

}
```

That's pretty uninspiring. We're going to need some fields in our data model to hold the various elements for each quote. Let's beef up our class to hold the content of the quote, the name of the author, and the date the entry was added, as shown in listing 1.8.

Listing 1.8 Our first domain class with teeth

```
class Quote {

   String content
   String author
   Date created = new Date()

   static constraints = {
   }
}
```

Now that we've got our data model, we need to go off and create our database schema, right? Wrong. Grails does all that hard work for you behind the scenes. Based on the definitions of the types in listing 1.8, and by applying some simple conventions, Grails creates a quote table, with `varchar` fields for the strings, and `Date` fields for the date. The next time we run `grails run-app`, our data model will be created on the fly.

But how will it know which database to create the tables in? It's time to configure a data source.

1.4.1 Configuring the data source

Grails ships with an in-memory database out of the box, so if you do nothing, your data will be safe and sound in volatile RAM. The idea of that makes most programmers a little nervous, so let's look at how we can set up a database that's a little more persistent.

In your /grails-app/conf/ directory, you'll find a file named DataSource.groovy. This is where you define the data source (database) that your application will use—you can define different databases for your development, test, and production environments. When you run `grails run-app` to start the local web server, it uses your development data source. Listing 1.9 shows an extract from the standard DataSource file, which shows the default data source.

Listing 1.9 Data source definition—in memory

```
development {
   dataSource {
      dbCreate = "create-drop"
      url = "jdbc:hsqldb:mem:devDB"
   }
}
```

Recreates database on every run

Specifies an in-memory database

We have two issues here. The first is that the `dbCreate` strategy tells Grails to drop and re-create your database on each run. This is probably not what you want, so let's change that to `update`, so Grails knows to leave our database table contents alone between runs (but we give it permission to add columns if it needs to).

The second issue relates to the URL—it's using an HSQLDB in-memory database. That's fine for test scripts, but not so good for product development. Let's change it to a file-based version of HSQLDB so we have some real persistence.

Our updated file is shown in listing 1.10.

Listing 1.10 Data source definition—persistent

```
development {
   dataSource {
      dbCreate = "update"
      url = "jdbc:hsqldb:file:devDB;shutdown=true"
   }
}
```

Preserves tables between runs

Specifies file-based database

Now we have a database that's persisting our data, so let's look at how we can populate it with some sample data.

1.4.2 Exploring database operations

We haven't done any work on our user interface yet, but it would be great to be able to save and query entries in our quotes table. To do this for now, we'll use the Grails console—a small GUI application that will start your application outside of a web server and give you a console to issue Groovy commands.

You can use the `grails console` command to tinker with your data model before your app is ready to roll. When we issue this command, our QOTD Grails application is bootstrapped, and the console GUI appears, waiting for us to enter some code. Figure 1.8 shows saving a new quote to the database via the console.

For our first exploration of the data model, it would be nice to create and save some of those `Quote` objects. Type the following into the console window, and then click the Run button (at the far right of the toolbar):

```
new Quote(author: 'Larry Wall',
    content: 'There is more than one method to our madness.').save()
```

The bottom half of the console will let you know you're on track:

```
Result: Quote : 1
```

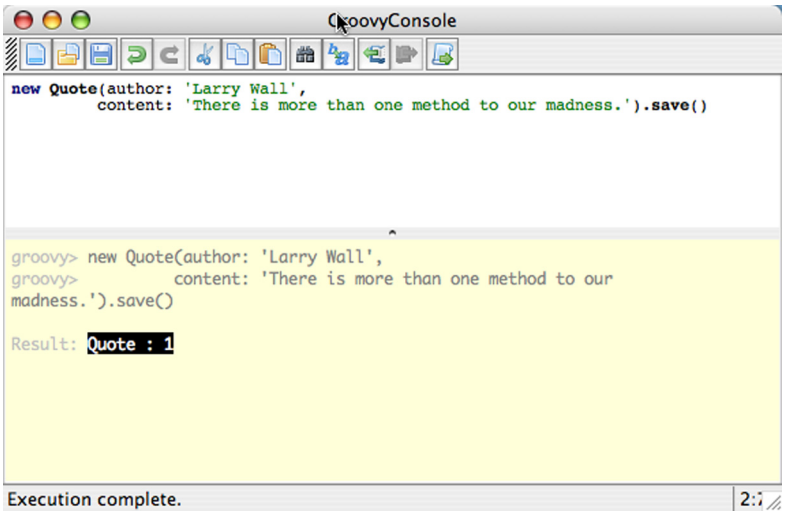

Figure 1.8 The Grails console lets your run commands from a GUI.

Where did that `save()` routine come from? Grails automatically endows domains with certain methods. Let's add a few more entries, and we'll get a taste of querying:

```
new Quote(author: 'Chuck Norris Facts', content: 'Chuck Norris always uses his
     own design patterns, and his favorite is the Roundhouse Kick').save()
new Quote(author: 'Eric Raymond', content: 'Being a social outcast helps you
     stay concentrated on the really important things, like thinking and
     hacking.').save()
```

Let's use another one of those dynamic methods (`count()`) to make sure that our data is being saved to the database correctly:

```
println Quote.count()
3
```

Looks good so far. It's time to roll up our sleeves and do some querying on our Quote database. To simplify database searches, Grails introduces special query methods on your domain class called *dynamic finders*. These special methods utilize the names of fields in your domain model to make querying as simple as this:

```
def quote = Quote.findByAuthor("Larry Wall")
println quote.content
There is more than one method to our madness.
```

Now that we know how to save and query, it's time to start getting our web application up and running. Exit the Grails console, and we'll learn a little about getting those quotes onto the web.

1.5 Adding UI actions

Let's get something on the web. First, we'll need an action on our `QuoteController` to return a random quote from our database. We'll work out the random selection

later—for now, let's cut some corners and fudge our sample data, as shown in listing 1.11.

Listing 1.11 Random refactored

```
def random = {
   def staticQuote = new Quote(author: "Anonymous",
         content: "Real Programmers Don't eat quiche")
   [ quote : staticQuote]
}
```

We'll also need to update our /grails-app/views/quote/random.gsp file to use our new `Quote` object:

```
<q>${quote.content}</q>
<p>${quote.author}</p>
```

There's nothing new here, just a nicer data model. This would be a good time to refresh your browser and see our static quote being passed through to the view. Give it a try to convince yourself it's all working.

Now that you have a feel for passing model objects to the view, and now that we know enough querying to be dangerous, let's rework our action in listing 1.12 to implement a real random database query.

Listing 1.12 A database-driven random

```
def random = {

   def allQuotes = Quote.list()
   def randomQuote
   if (allQuotes.size() > 0) {
      def randomIdx = new Random().nextInt(allQuotes.size())
      randomQuote = allQuotes[randomIdx]
   } else {
      randomQuote = new Quote(author: "Anonymous",
         content: "Real Programmers Don't eat Quiche")
   }
   [ quote : randomQuote]

}
```

1 **Obtains list of quotes**

2 **Selects random quote**

3 **Generates default quote**

4 **Passes quote to the view**

With our reworked `random` action, we're starting to take advantage of some real database data. The `list()` method 1 will return the complete set of `Quote` objects from the quote table in the database and populate our `allQuotes` collection. If there are any entries in the collection, we select a random one 2 based on an index into the collection; otherwise, we use a static quote 3. With all the heavy lifting done, we return a `randomQuote` object to the view in a variable called `quote` 4, which we can access in the GSP file.

Now that we've got our random feature implemented, let's head back to http://localhost:8080/qotd/quote/random to see it in action. Figure 1.9 shows our random feature in action.

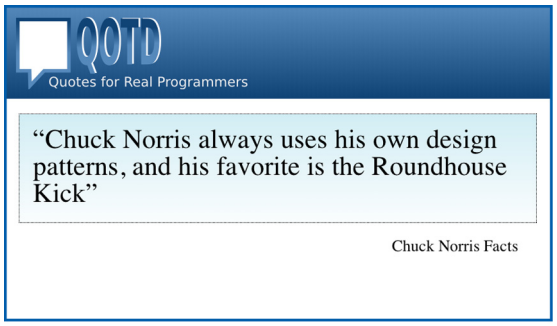

Figure 1.9 Our random quote feature in action

1.5.1 Scaffolding: just add rocket fuel

We've done all the hard work of creating our data model. Now we need to enhance our controller to handle all the CRUD actions to let users put their own quotes in the database.

That's if we want to do a slick job of it. But if we want to get up and running quickly, Grails offers us a fantastic shortcut called *scaffolding*. Scaffolds dynamically implement basic controller actions and views for the common things you'll want to do when CRUDing your data model.

How do we scaffold our screens for adding and updating quote-related data? It's a one-liner for the `QuoteController`, as shown in listing 1.13.

Listing 1.13 Enabling scaffolding

```
class QuoteController {
   def scaffold = true
  // our other stuff here...
}
```

That's it. When Grails sees a controller marked as `scaffold = true`, it goes off and creates some basic controller actions and GSP views on the fly. If you'd like to see it in action, head over to http://localhost:8080/qotd/quote/list and you'll find something like the edit page shown in figure 1.10.

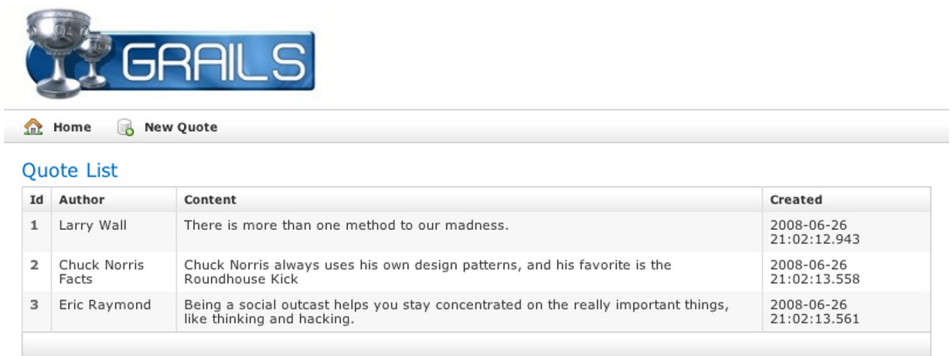

Figure 1.10 The `list()` scaffold in action

Figure 1.11 Adding a quote has never been easier.

Click on the New Quote button, and you'll be up and running. You can add your new quote as shown in figure 1.11.

That's a lot of power to get for free. The generated scaffolds are probably not tidy enough for your public-facing sites, but they're absolutely fantastic for your admin screens and perfect for tinkering with your database during development (where you don't want the overhead of mocking together a bunch of CRUD screens).

1.5.2 Surviving the worst case scenario

Our model is looking good and our scaffolds are great, but we're still missing some pieces to make things a little more robust. We don't want users putting dodgy stuff in our database, so let's explore some validation.

Validation is declared in our `Quote` object, so we just need to populate the `constraints` closure with all the rules we'd like to apply. For starters, let's make sure that users always provide a value for the author and content fields, as shown in listing 1.14.

Listing 1.14 Adding basic validation

```
class Quote {

   String content
   String author
   Date created = new Date()

   static constraints = {
         author(blank:false)
         content(maxSize:1000, blank:false)
   }

}
```

Enforces data validation

These constraints tell Grails that neither `author` nor `content` can be blank (neither `null` nor `0` length). If we don't specify a size for `String` fields, they'll end up being defined `VARCHAR(255)` in our database. That's probably fine for `author` fields, but our content may expand on that a little. That's why we've added a `maxSize` constraint.

Entries in the `constraints` closure also affect the generated scaffolds. For example, the ordering of entries in the `constraints` closure also affects the order of the fields in generated pages. Fields with constraint sizes greater than 255 characters are rendered as `HTML TEXTAREA`s rather than `TEXT` fields. Figure 1.12 shows how error messages display when constraints are violated.

Figure 1.12 When constraints are violated, error messages appear in red.

1.6 Improving the architecture

Spreading logic across our controller actions is all well and good. It's pretty easy to track down what goes where in our small app, and maintenance isn't a concern right now. But as our quotation app grows, we'll find that things get a little more complex. We'll want to reuse logic in different controller actions, and even across controllers. It's time to tidy up our business logic, and the best way to do that in Grails is via a service.

Let's create our service and learn by doing:

```
grails create-service quote
```

This command creates a skeleton quote service in /grails-app/services/QuoteService.groovy:

```
class QuoteService {

   boolean transactional = true

   def serviceMethod() {

   }
}
```

You'll notice that services can be marked transactional—more on that later. For now, let's move our random quote business logic into its own method in the service, as shown in listing 1.15.

Listing 1.15 Beefing up our service

```
class QuoteService {

   boolean transactional = false

   def getStaticQuote() {
      return new Quote(author: "Anonymous",
         content: "Real Programmers Don't eat quiche")
   }

   def getRandomQuote() {

      def allQuotes = Quote.list()
      def randomQuote = null
      if (allQuotes.size() > 0) {
         def randomIdx = new Random().nextInt(allQuotes.size())
         randomQuote = allQuotes[randomIdx]
      } else {
         randomQuote = getStaticQuote()
      }
      return randomQuote

   }
}
```

Now our service is implemented. How do we use it in our controller? Again, conventions come into play. We just add a new field to our controller called `quoteService`, and Grails will inject the service into the controller. Listing 1.16 shows the updated code.

Listing 1.16 Invoking our service

```
class QuoteController {

   def scaffold = true

   def quoteService

   def random = {
      def randomQuote = quoteService.getRandomQuote()
      [ quote : randomQuote ]
   }

}
```

Doesn't that feel much tidier? Our `QuoteService` looks after all the business logic related to quotes, and our `QuoteController` helps itself to the methods it needs. If you have experience with Inversion of Control (IoC) containers, such as Spring or Google Guice, you will recognize this pattern of application design as Dependency Injection (DI). Grails takes DI to a whole new level by using the convention of variable names to determine what gets injected. But we have yet to write a test for our business logic, so now's the time to explore Grails' support for testing.

1.6.1 Your first Grails test case

Testing is a core part of today's agile approach to development, and Grails' support for testing is wired right into the framework. Grails is so insistent about testing that when we created our `QuoteService`, Grails automatically created a shell unit-test case in /grails-app/test/unit/QuoteServiceTests.groovy to encourage us to do the right thing. But unit tests (which we'll explore in chapter 7) require a bit of mock trickery to simulate database calls. For now, we want an integration test (which gives us a "real" in-memory database to test against). We create one of those with this command:

```
grails create-integration-test QuoteServiceIntegration
```

This will tell Grails to create a shell /grails-app/test/integration/QuoteServiceIntegrationTests.groovy file. We've given the test an "IntegrationTests" suffix to make sure its class name doesn't clash with our existing unit test in /test/unit/QuoteServiceTests.groovy. Listing 1.17 shows what the initial integration test looks like.

Listing 1.17 Our first test case

```
import grails.test.*

class QuoteServiceTests extends GrailsUnitTestCase {

   protected void setUp() {
      super.setUp()
   }

   protected void tearDown() {
      super.tearDown()
   }

   void testSomething() {

   }
}
```

It's not much, but it's enough to get us started. The same Convention over Configuration rules apply to tests, so let's beef up our `QuoteServiceIntegrationTests` case to inject the service that's under test, as shown in listing 1.18.

Listing 1.18 Adding real tests

```
class QuoteServiceTests extends GrailsUnitTestCase {

   def quoteService

   void testStaticQuote() {
      def staticQuote = quoteService.getStaticQuote()
      assertEquals("Anonymous", staticQuote.author)
      assertEquals("Real Programmers Don't eat Quiche", staticQuote.content)
   }
}
```

There's not too much that can go wrong with the `getStaticQuote()` routine, but let's give it a workout for completeness. To run your tests, execute `grails test-app`. You should see something like the results in listing 1.19.

Listing 1.19 Test output

```
-------------------------------------------------------
Running 1 Integration Tests...
Running test QuoteServiceIntegrationTests...
              testStaticQuote...SUCCESS
Integration Tests Completed in 284ms
-------------------------------------------------------
```

Listing 1.19 shows us that our tests are running fine. Grails also generates an HTML version of our test results, which you can view by opening /grails-app/test/reports/html/index.html in a web browser. From there you can browse the whole project's test results visually and drill down into individual tests to see what failed and why, as shown in figure 1.13.

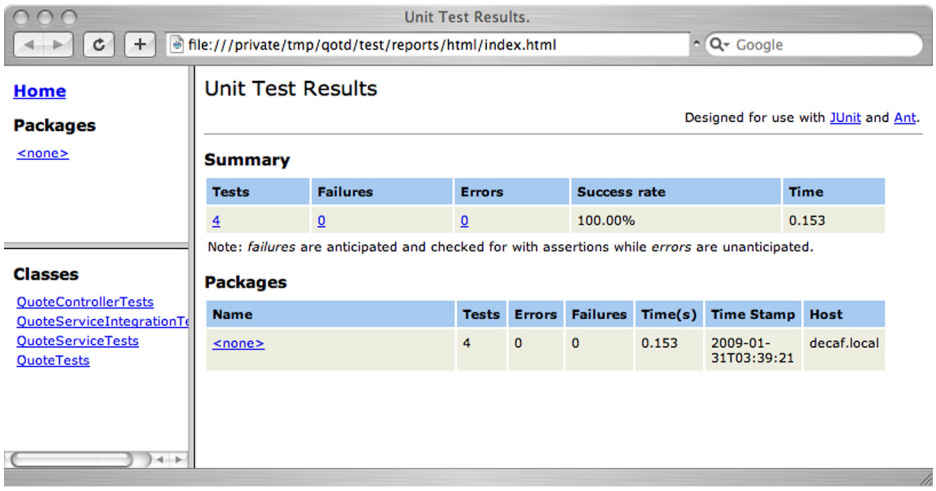

Figure 1.13 HTML reports from the integration test run

We'll learn how to amp up our test-coverage in chapter 7, but for now we have a test up and running, and we know how to view the output.

1.6.2 *Going Web 2.0: Ajax-ing the view*

Our sample application wouldn't be complete without adding a little Ajax (Asynchronous JavaScript and XML) secret sauce to spice things up. If you haven't heard much about Ajax, it's a way of updating portions of a web page using JavaScript. By using a little Ajax, we can make our web application a lot more responsive by updating the quote without having to reload the masthead banners and all our other page content. It also gives us a chance to look at Grails tag libraries.

Let's Ajax-ify our random.gsp view. First, we have to add the Ajax library to our `<head>` element (we'll use Prototype, but Grails also lets you use YUI, Dojo, or others). An updated portion of random.gsp is shown in listing 1.20.

Listing 1.20 Adding a JavaScript library for Ajax

```
<head>
   <title>Random Quote</title>
   <g:javascript library="prototype" />
</head>
```

Then, in the page body of random.gsp, we'll add a menu section that allows the user to display a new quote or head off to the admin screens. We'll use Grails' taglibs to create both our Ajax link for refreshing quotes and our standard link for the admin interface. Listing 1.21 shows our new menu HTML. We'll add this snippet before the `<div>` tag that hosts the body of the page.

Listing 1.21 Invoking Ajax functionality

```
<ul id="menu">
   <li>
      <g:remoteLink action="ajaxRandom" update="quote">
         Next Quote
      </g:remoteLink>
   </li>
   <li>
      <g:link action="list">
         Admin
      </g:link>
   </li>
</ul>
```

You've seen these sorts of tag library calls earlier in the chapter (in section 1.3.3), where we used them to generate a standardized layout for our application. In this example, we introduce a `g:remoteLink`, Grails' name for an Ajax hyperlink, and `g:link`, which is the tag for generating a standard hyperlink.

When you click on this link, Grails will call the `ajaxRandom` action on the controller that sent it here—in our case, the `QuoteController`—and will place the returned HTML inside the div that has an ID of `quote`. But we haven't written our `ajaxRandom` action, so let's get to work. Listing 1.22 shows the updated fragment of QuoteController.groovy with the new action.

Listing 1.22 The server side of Ajax

```
def ajaxRandom = {
   def randomQuote = quoteService.getRandomQuote()
   render "<q>${randomQuote.content}</q>" +
                     "<p>${randomQuote.author}</p>"
}
```

We'd already done the heavy lifting in our quote service, so we can reuse that here. Because we don't want our Grails template to decorate our output, we're going to write our response directly to the browser (we'll talk about more elegant ways of doing this in later chapters). Let's take our new Ajax app for a spin, as shown in figure 1.14.

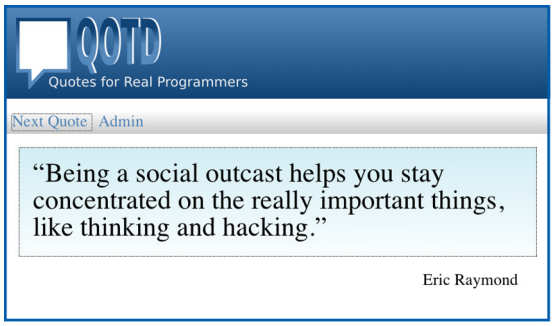

Figure 1.14 Our Ajax view in action

To convince yourself that all the Ajax snazziness is in play, click on the Next Quote menu item a few times. Notice how there's no annoying repaint of the page? You're living the Web 2.0 dream.

1.6.3 Bundling the final product: creating a WAR file

Look how much we've achieved in half an hour! But it's no good running the app on your laptop—you need to set it free and deploy it to a real server out there in the cloud. For that, you'll need a WAR file, and Grails makes creating one a one-liner:

```
grails war
```

Watch the output, and you'll see Grails bundling up all the JARs it needs, along with your Grails application files, and creating the WAR file in your project's root directory:

```
Done creating WAR /Users/glen/qotd/qotd-0.1.war
```

Now you're ready to deploy.

1.6.4 And 55 lines of code later

We've learned a lot. And we've coded a fair bit too. But don't take my word for it; let's let Grails crunch the numbers for us with a `grails stats` command. Listing 1.23 shows the `grails stats` command in action.

Listing 1.23 Crunching numbers: the `grails stats` command in action

```
grails stats

      +----------------------+-------+-------+
      | Name                 | Files |  LOC  |
      +----------------------+-------+-------+
      | Controllers          |     1 |    13 |
      | Domain Classes       |     1 |     9 |
      | Services             |     1 |    17 |
      | Integration Tests    |     3 |    16 |
      +----------------------+-------+-------+
      | Totals               |     6 |    55 |
      +----------------------+-------+-------+
```

Only 55 lines of code (LOC)! Maybe we haven't coded as much as we thought. Still, you'd have to say that 55 lines isn't too shabby for an Ajax-powered, user-editable, random quote web application.

That was quite an introduction to Grails. We've had a taste of models, views, controllers, services, taglibs, layouts, and unit tests. And there's much more to explore. But before we go any further, it might be good to explore a little Groovy.

1.7 Summary and best practices

Congratulations, you've written and deployed your first Grails app, and now you have a feel for working from scratch to completed project. The productivity rush can be quite addictive.

Here are a few key tips you should take away from this chapter:

- *Rapid iterations are key.* The most important take-away for this chapter is that Grails fosters rapid iterations to get your application up and running in record time, and you'll have a lot of fun along the way.
- *Noise reduction fosters maintenance and increases velocity.* By embracing Convention over Configuration, Grails gets rid of tons of XML configuration that used to kill Java web frameworks.
- *Bootstrapping saves time.* For the few cases where you do need scaffolding code (for example, in UI design), Grails generates all the shell boilerplate code to get you up and running. This is another way Grails saves you time.
- *Testing is inherent.* Grails makes writing test cases easy. It even creates shell artifacts for your test cases. Take the time to learn Grails' testing philosophy (which we'll look at in depth in chapter 7) and practice it in your daily development.

There's certainly a lot more to learn. We'll spend the rest of the book taking you through all the nuts and bolts of developing full-featured, robust, and maintainable web apps using Grails, and we'll point out the tips, tricks, and pitfalls along the way.

JAVA

Grails IN ACTION

Glen Smith and Peter Ledbrook, FOREWORD BY Dierk König

Free ebook
SEE INSERT

Web apps shouldn't be hard to build, right? The developers of Grails agree. This hyper-productive open-source web framework lets you "code by convention," leaving you to focus on what makes your app special. Through its use of Groovy, it gives you a powerful, Java-like language and full access to all Java libraries. And you can adapt your app's behavior at runtime without a server restart.

Grails in Action is a comprehensive guide to the Grails framework. First, the basics: the domain model, controllers, views, and services. Then, the fun! Dive into a Twitter-style app with features like AJAX/JSON, animation, search, wizards—even messaging and Jabber integration. Along the way, you'll discover loads of great plugins that'll make your app shine. Learn to integrate with existing Java systems using Spring and Hibernate. You'll need basic familiarity with Java and the web. Prior experience with Groovy is not necessary.

What's Inside

- A concise Groovy primer
- Advanced UI development
- Enterprise integration
- Plugin development
- Tips and tricks from the trenches

About the Authors

A frequent speaker and the co-host of the Grails podcast, **Glen Smith** launched the first public-facing Grails app (an SMS Gateway) on Grails 0.2. **Peter Ledbrook** is a core Grails developer and author of several popular plugins, who has worked as an engineer for both G2One and SpringSource.

For online access to the authors, code samples, and a free ebook for owners of this book, go to www.manning.com/GrailsinAction

"... rock-solid solutions delivered with ease."
—From the Foreword by Dierk König

"Bulging with serious content and tips. Bursting with fun."
—Paul King, ASERT

"Excellent resource, both as a guide and a reference."
—John Guthrie, Sybase

"Grails is the PHP of Web 2.0 and this book is pure gold."
—John G. Ledo
Ledo Works Inc.

"... an approachable and entertaining book ... sure to simplify your Grails development."
—Joe McTee, JEKLsoft

"Great framework + Great authors = Awesome book!"
—Dave Klein, Contegix

"Read this book if you want to raise your Grails IQ by 100+ points!"
—Zan Thrash
Thrash Consulting

ISBN-13: 978-1933988931
ISBN-10: 1933988932
54499
9 781933 988931

MANNING US $44.99/CAN $56.99 [INCLUDES EBOOK]