Grails IN ACTION

SECOND EDITION

Glen Smith
Peter Ledbrook

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Grails in Action Second Edition
version 15

brief contents

Part 1
Introducing Grails

The field of Java-based web application frameworks has made great strides in usability, but creating a new application with them is still hard work. Grails' core strength is developing web applications quickly, so you'll jump into writing your first application right away.

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

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

By the end of part 1, you'll understand 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
- Key components of a Grails application
- Developing and deploying your first Grails application

"Help, I've lost my Mojo!" That statement is a concise summary of what developers feel when working with one of the plethora of Java web frameworks. Each change requires time spent editing configuration files, customizing web.xml files, writing injection definitions, tweaking build scripts, modifying page layouts, and restarting apps. Aaaahhhh! "Where has all the fun gone? Why is everything so tedious? I wanted to whip up a quick app to track our customer signups! There must 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 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 thinking, "Please don't let this be YAJWF (Yet Another Java Web Framework)!" Because if the Java development world is famous for one thing, it's having an unbelievably large number of web frameworks available. Struts, WebWork, JavaServer Faces (JSF), Spring MVC, Seam, Wicket, Tapestry, Stripes, Google Web Toolkit (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 excited about Grails because we think it's time that Java web app development was fun again! It's time you 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 take you through developing your first Grails app. Not a toy, either. Something you can deploy and show your friends. An app that's data-driven and Ajax-powered that has full CRUD (create, read, update, delete) implementation, a template-driven layout, and even unit tests. In the time it takes to eat your lunch, with less than 100 lines of code. Seriously.

But before you fire up your IDE and get your hands dirty writing code, you may need more convincing about why Grails is such a game-changer and should be on your radar.

1.1 Introducing Grails

Grails is a next-generation Java web development framework that generates 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 Why Grails changed the game

Grails entered the Java Web Application landscape in 2006 and has grown steadily in adoption ever since. 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 such as `Book.findByTitle("Grails in Action")` or `Book.findAllBy-DatePublishedGreaterThanAndTitleLike(myDate, "Grails")`, even though none of those methods existed on the `Book` object.

Even better, you could 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 and used by enterprise Java developers. 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 behind any of their existing skills, libraries, or infrastructure.

1.1.2 Seven big ideas

That's enough history about how Grails came to be such a popular Java web framework. But if you (or your manager) need further convincing that 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 drove Grails to such a dominant position in the emerging next-gen Java web frameworks market.

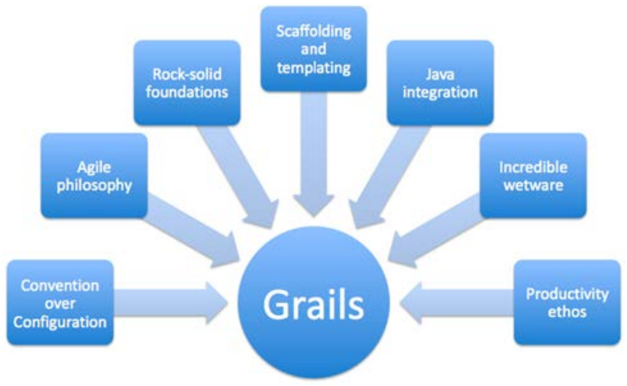


Figure 1.1 The Grails ecosystem is a powerful confluence of people, ideas, and technology.

BIG IDEA #1: CONVENTION OVER CONFIGURATION

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

- Add a controller class called `ShopController` 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 link up everything 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.

Grails is about convention *over* configuration, not convention *instead of* configuration. If you need to tweak the defaults, the power is there for you. Grails makes overriding the defaults easy, and you 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.

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 a real pain in Java a complete joy. Whether it's processing form posts, implementing tag libraries, or writing test cases, Grails offers 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 restarting 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 when 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.

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 use today, and for good reason: they're reliable and battle-tested.

Building on Spring and Hibernate also means there's no new 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 `Application-Context`). None of your learning time on Spring and Hibernate is wasted.

It doesn't matter if you're new to Grails and don't have a background in Spring and Hibernate. Few Grails development cases fall back to that level, but know 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 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 entire Groovy and Grails stack. In recent times, SpringSource was acquired by VMWare and spun off into a dedicated Spring-related development and support organization called Pivotal (http://gopivotal.com/). This has also introduced a range of support options to the platform that are useful to organizations looking for 24/7 Groovy and Grails support backup.

BIG IDEA #4: SCAFFOLDING AND TEMPLATING

If you've ever tried bootstrapping a Spring MVC application by hand, you know it isn't pretty. You need a directory of JAR files, bean definition files, web.xml customizations, annotated POJOs, Hibernate configuration files, database-creation script, and a build system to turn it all into a running application. It's hard work, and you may 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 run in your browser. All the same stuff happens 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 skeleton for you (along with a skeleton test case). The same goes for views, services, domain classes, and all 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 CRUD operations without a single line of code.

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 problems and design solutions are 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 and don't want to leave behind the amazing selection of Java libraries we know and love. Whether that's in-house data transfer objects (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 verbose XML configuration files. And as we've said before, you can reuse your Hibernate mappings and Spring resource files if you're so inclined!

Big idea #6: incredible COMMUNITY

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:

- Groovyblogs.org aggregates what's happening in the Groovy and Grails blogosphere and is full of interesting articles.
- Sites such as Facebook and LinkedIn host Grails social networking options.
- A Grails podcast (grailspodcast.com) 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. You'll find literally hundreds of time-saving plugins (and in chapter 10, we introduce you to the most popular ones).

BIG IDEA #7: PRODUCTIVITY ETHOS

Grails is about more than building web applications—it's about executing your vision quickly so that you can get on to more important "life stuff": hanging out with your family, walking your dog, learning rock guitar, or getting your veggie patch growing big zucchinis. Web apps come and go; zucchinis are forever. Grails productivity gives you that sort of sage-like perspective.

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 in one day work that used to take two weeks, you start to feel good about going home early. Working with such a productive framework even makes your hobby time more fun. You can complete all those Web 2.0 startup website ideas you've dreamed about, but that ended up as half-written Struts or Spring MVC apps. 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

To get Grails up and running, review the installation process shown in figure 1.2.

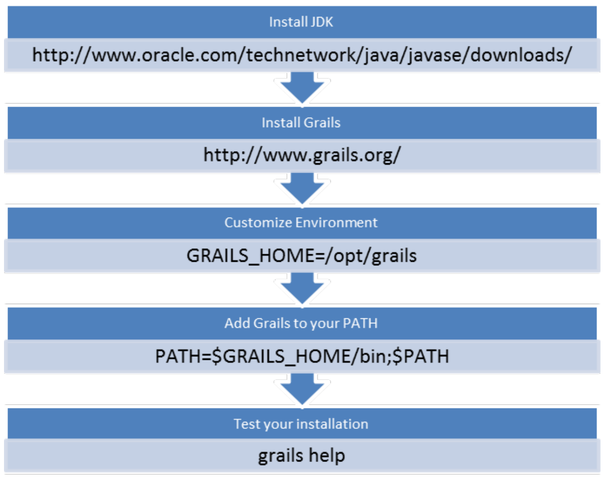


Figure 1.2 The Grails installation process

1. Install a JDK (version 1.6 or later).

   Run `javac -version` from your command prompt to verify the version you have. Most PCs come with Java preinstalled these days, so you may not need this step.

2. After your JDK is installed, download the latest Grails distro from www.grails.org and unzip it to your favorite installation area.
3. 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, edit the ~/.profile script to contain lines such as these:

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

   On Windows, go into System Properties to define `GRAILS_HOME` and update your `PATH` setting.
4. To verify that Grails is installed correctly, run `grails help` from the command line.

   This will give you a handy list of Grails commands and confirm that everything is running as expected.

Note on Grails versions

The book is based on Grails 2.3.7, but the latest version of Grails may be different by the time you read this. The best way to ensure that you're running the correct version of Grails with all our sample code is via the Grails wrapper:

```
./grailsw <command>
```

You don't need the starting ./ on Windows, it's only for Unix-like systems.

New projects created by Grails 2.3 and above already contain the wrapper, but for Grails 2.1 and 2.2 you need to explicitly run `grails wrapper` if you want it for your own projects.

As your Grails applications become more sophisticated, you'll want to take advantage of the fantastic Grails IDE support available. You can find Grails plugin support for your preferred IDE—IntelliJ, NetBeans, or Eclipse. We won't develop much code in this chapter, so a text editor is all you need. Fire up your favorite editor, and let's talk about your sample application.

1.3 QOTD: your sample program

If you're writing a small application, you may as well have fun. This example is a Quote-of-the-Day web application in which you'll capture and display famous programming quotes from development rock stars throughout time. You'll let the user add, edit, and cycle through programming quotes, and add some Ajax sizzle to give it a modern feel. You'll want a short URL for your application, so make qotd your application's working title.

NOTE You can download the sample apps for this book, including CSS and associated graphics, from the book's site (www.manning.com/gsmith2). To view the latest issues and check out the latest sources, see the GitHub project (https://github.com/GrailsInAction/graina2) for details.

It’s time to start your world-changing 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 your application files. Change to that directory now, which is where you’ll stay for the rest of the chapter.

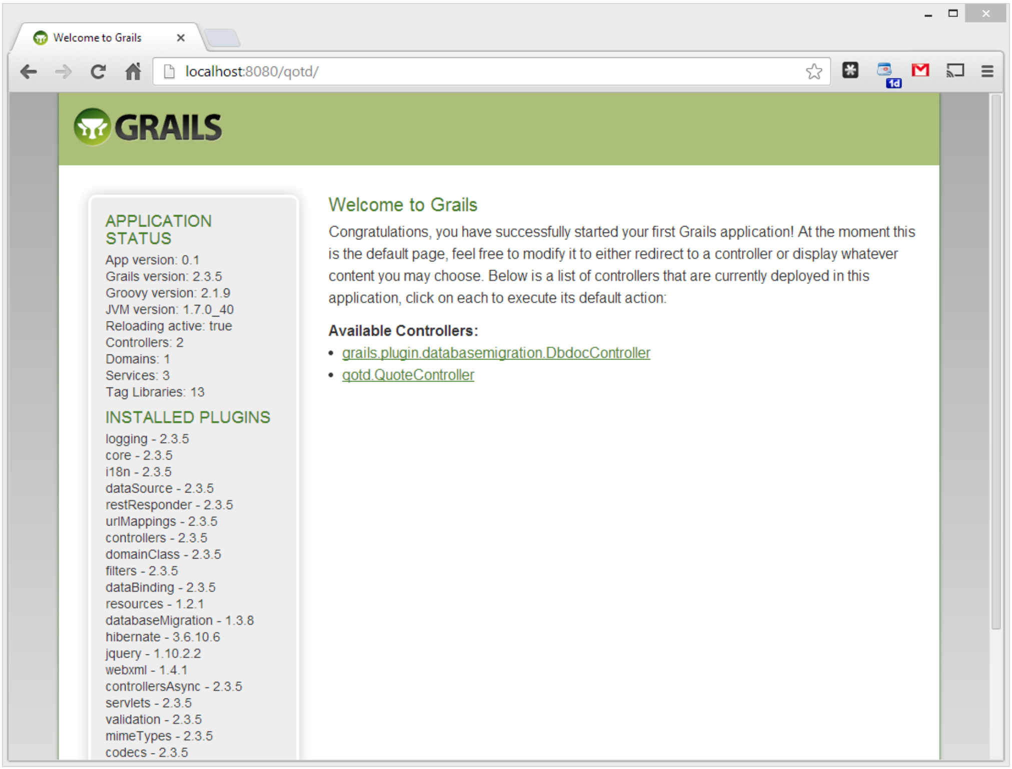
Because you’ve done the hard work of building the application, it’d be a shame not to enjoy the fruit of your labor. To run the app, enter:

```
grails run-app
```

Grails ships with a Tomcat plugin used to host your application during the development and testing life cycle. When you run the `grails run-app` - command, Grails compiles and starts 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 shows your QOTD application up and running in a browser.


Welcome to Grails

localhost:8080/qotd/

GRAILS

APPLICATION STATUS

App version: 0.1
Grails version: 2.3.5
Groovy version: 2.1.9
JVM version: 1.7.0_40
Reloading active: true
Controllers: 2
Domains: 1
Services: 3
Tag Libraries: 13

INSTALLED PLUGINS

logging - 2.3.5
core - 2.3.5
i18n - 2.3.5
dataSource - 2.3.5
restResponder - 2.3.5
urlMappings - 2.3.5
controllers - 2.3.5
domainClass - 2.3.5
filters - 2.3.5
dataBinding - 2.3.5
resources - 1.2.1
databaseMigration - 1.3.8
hibernate - 3.6.10.6
jquery - 1.10.2.2
webxml - 1.4.1
controllersAsync - 2.3.5
servlets - 2.3.5
validation - 2.3.5
mimeTypes - 2.3.5
codecs - 2.3.5

Welcome to Grails

Congratulations, you have successfully started your first Grails application! At the moment this is the default page, feel free to modify it to either redirect to a controller or display whatever content you may choose. Below is a list of controllers that are currently deployed in this application, click on each to execute its default action:

Available Controllers:

- grails.plugin.databasemigration.DbdocController
- qotd.QuoteController

Figure 1.3 Your app is up and running.

After you've taken in the home page, you can stop the application by pressing `Ctrl-C` or running grails stop-app from another terminal/command prompt. Alternatively, you can leave the application running and issue Grails commands from a separate terminal/command prompt in your operating system.

Running on a custom port (not 8080)

If port 8080 isn't for you (because you have another process, such as Tomcat, running), 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 learn more about these files in chapter 18.

1.3.1 Writing a controller

You have your application built and deployed, but you're short on an engaging user experience. Before you go further, now is a good time to learn that Grails handles interaction with users 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 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 respond with a list of the artifacts it generated:

```
| Created file grails-app/controllers/qotd/QuoteController.groovy
| Created file grails-app/views/quote
| Created file test/unit/qotd/QuoteControllerSpec.groovy
```

A word on package naming

If you omit the package name for a Grails artifact, it will default to the name of the app (in the example above, if you do a `grails create-controller quote`, it creates an artifact called /grails-app/qotd/QuoteController.groovy).

For production code, the Grails community has settled on the standard Java-based convention where your artifacts should be created with your org domain name. Grails lets you change the default package name for your app in /grails-app/conf/Config.groovy. For this chapter's example, you might choose to change the setting in that file to read:

```
grails.project.groupId = "com.grailsinaction.qotd"
```

With such a setting in play, when you do `grails create-controller quote` it will create the class in /grails-app/controller/com/grailsinaction/qotd/QuoteController.groovy. It's a great key saver change to make at the start of a new Grails project- however to prevent surprises for people picking up this chapter halfway through, we're just going to stick with the default package name of qotd for now.

Grails creates this skeleton controller in /grails-app/controllers/qotd/QuoteController.groovy. You'll notice that Grails sorted out the capitalization for you. Here is the skeleton:

```
package qotd

class QuoteController {
    def index() { }
}
```

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

```
package qotd

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 you dip your toes into Ajax waters, but for now let's use it to deliver your "Real Programmers" heading.

How do you invoke your 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 the convention over configuration pattern comes in.

Ruby on Rails introduced the idea that XML configuration (or configuration of any sort) can be avoided if the framework makes opinionated choices for you about how things fit together. Grails embraces the same philosophy. Because your controller is called `QuoteController`, Grails will expose its actions over the URL /qotd/quote/youraction. Figure 1.4 gives a visual breakdown of how URLs translate to Grails objects.

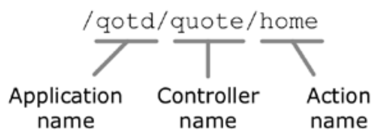
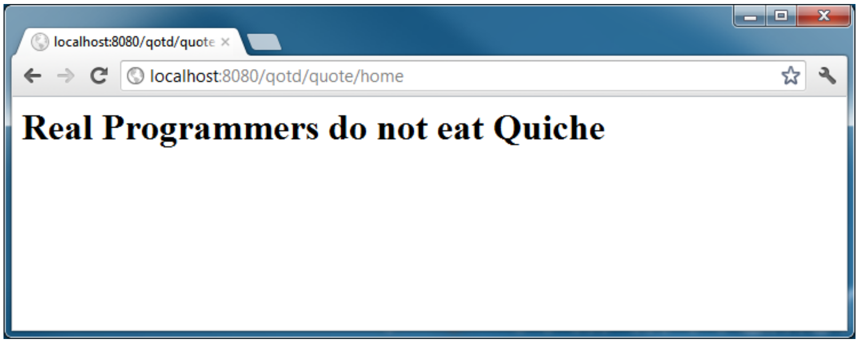


Figure 1.4 How URLs translate to Grails objects

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

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

Figure 1.5 shows your brand-new application up and running without a single line of XML.



Real Programmers do not eat Quiche

Figure 1.5 Adding your functionality

If you're wondering about that `index()` routine in the skeleton controller code, that's the method called when the user omits the action name. If you decide all references to /qotd/quote/ should end up at /qotd/quote/home, you need to tell Grails about that with a default action such as the one in the following listing.

Listing 1.1 Handling redirects

```
package qotd

class QuoteController {

    static defaultAction = "home"

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

The app looks good so far, but having that HTML embedded in your source is nasty. Now that you've learned about controllers, it's time to get acquainted with views.

1.3.2 Generating an HTML page: 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 to design the pages. The solution is to move your display logic to a separate file known as the *view*. 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 make use of Groovy Server Pages (GSP). The concepts are similar.

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

Begin by implementing your random action as shown in the following code. We'll handle the view next.

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

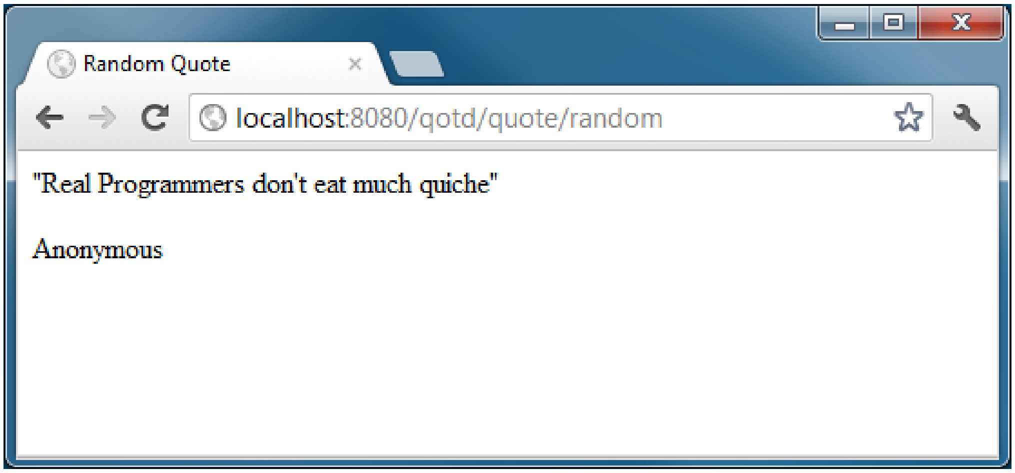
What's with all those square brackets? That's how the controller action passes information to the view. If you're an old-school servlet programmer, think of it as request-scoped data. The `[:]` operator in Groovy creates a `Map`, so you're passing a series of key/value pairs through to your view.

Where does your view fit into this, and where will you put your GSP file so that Grails can find it? Use the naming conventions you used for the controller, coupled with the name of your action, and place the GSP in /grails-app/views/quote/random.gsp. If you follow that pattern, no configuration is required.

Let's create a GSP file that references your `Map` data, as shown in the following code:

```
<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 worked with JSPs, it will be old news to you. If you haven't worked with JSPs, 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 1.6 shows your new markup in action.



"Real Programmers don't eat much quiche"

Anonymous

Figure 1.6 Your view in action

1.3.3 Adding style with Grails layouts

You've now written your piece of backend functionality, but the output isn't engaging—no gradients, no giant text, no rounded corners. Everything looks mid-90s.

You think it's time for CSS, but let's plan ahead. If you mark up random.gsp with CSS, you're going to have to add those links to the header of every page in the app. Grails has a better way: layouts.

Layouts give you a way to specify layout templates for certain parts of your application. For example, you may want all of the quote pages (random, by author, by date) styled with a common masthead and navigation links; only the body content should change. To do this, let's mark up your target page with IDs you can use for your CSS:

```
<html>
<head>
    <title>Random Quote</title>
</head>
<body>
    <div id="quote">
        <q>${content}</q>
        <p>${author}</p>
    </div>
</body>
</html>
```

Now, how do you apply those layout templates (masthead and navigation) we discussed earlier? Like everything else in Grails, layouts follow a convention over configuration style. To have all your `QuoteController` actions share the same layout, create a file called /grails-app/views/layouts/quote.gsp. Grails doesn't have shortcuts for layout creation, so you've got to roll this one by hand. The following listing shows your attempt at writing a layout.

Listing 1.2 Adding a layout

```
<html>
    <head>
        <title>QOTD &raquo; <g:layoutTitle/></title>                 #1
        <g:external dir="css" file="snazzy.css"/>                    #2
        <g:layoutHead />          #3
        <r:layoutResources />     #4
    </head>
    <body>
        <div id="header">
            <g:img dir="images" file="logo.png" alt="logo"/>
        </div>
        <g:layoutBody />           #5
    </body>
</html>
```

#1 Merges title from target page
#2 Creates relative link to CSS file
#3 Merges head elements from target page
#4 Merges in JavaScript, CSS, and other resources
#5 Merges body elements from target page

Let’s break down the use of angle brackets. Because this is a template page, the contents of your target page (random.gsp) will be merged with this template before you send any content back to the browser. Under the hood, Grails uses SiteMesh, the popular Java layout engine, to do the merging for you. Figure 1.7 shows the merge process.

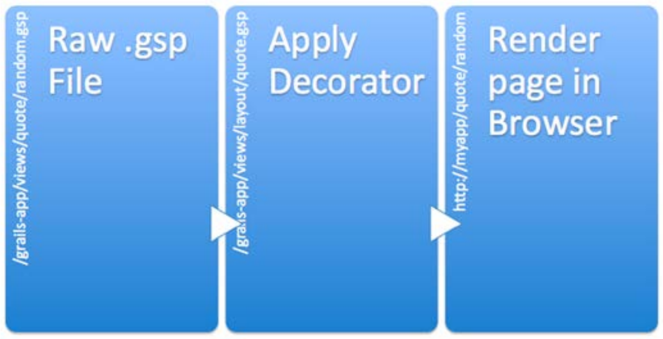

/grails-app/views/quote/random.gsp
Raw .gsp File
/grails-app/views/layout/quote.gsp
Apply Decorator
http://myapp/quote/random
Render page in Browser

Figure 1.7 SiteMesh decorates a raw GSP file with a standard set of titles and sidebars.

To make your layout template in listing 1.2 work, it needs a way to access elements of the target page (when you merge the title of the target page with the template, for example). It’s time to introduce you to taglibs because access is achieved through Grails’ template taglibs.

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 1.2, you took advantage of the `<g:external>`, `<g:layoutHead>`, and `<g:layoutBody>` tags. When the client’s browser requests the page, Grails replaces those tag calls with real HTML, and the contents of the HTML will depend on what the individual tag generates. For instance, that `<g:external>` tag #2 will generate an HTML `<link>` element that points to the URL for snazzy.css.

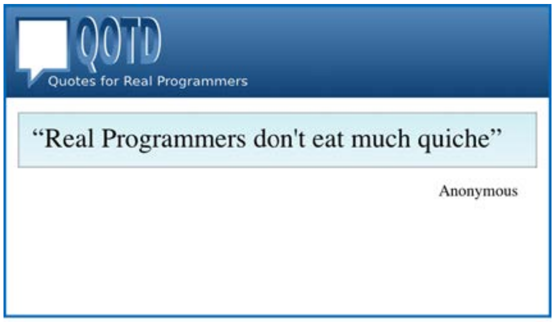
In the title block of the page, you include your QOTD title and then follow it with chevrons (>>) represented by the HTML character code `&raquo;`, and then add the title of the target page itself #1.

After the rest of the head tags, you use a `<g:layoutHead>` tag to merge the contents of the HEAD section of any target page #3. 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.

With your head metadata in place, it's time to lay out any other HEAD-bound resources that your page might need in the head section with a `<g:layoutResources>` tag #4. (This is any other CSS or JavaScript that the Grails resources infrastructure requires in the HEAD section of this page. More on this magic in the Advanced UI chapter!.

Finally, you get to the body of the page. You output your common masthead div to get your Web 2.0 gradient and cute icons, and then you call `<g:layoutBody>` to render the BODY section of the target page #5.

Refresh your browser to see how you're doing. Figure 1.8 shows your styled page.



"Real Programmers don't eat much quiche"

Anonymous

Figure 1.8 QOTD with some funky CSS skinning

Getting the CSS and artwork

If you're following along step-by-step at your workstation, you'll be keen to grab the CSS and image files that go along with the styling shown previously (so your local app can look the same). You can grab the few files you need (/web-app/css/snazzy.css and /web-app/images/ directly from the chapter 1 source code available for download from www.manning.com/gsmith2 or directly from the current source code on GitHub (https://github.com/GrailsInAction/graina2).

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

Fantastic. You're up and running with a controller, view, and template. But things are still static in the data department. You're overdue to learn how Grails handles information in the database. When you have that under your belt, you can circle back and implement a real random action.

1.4 Creating the domain model

You've begun your application, and you can deploy it to your testing web container. But let's not overstate your progress—Google isn't about to buy you yet. Your app lacks a certain pizzazz. It's time to add interactivity allowing users to add new quotations to the database. To store those quotations, you'll need to learn how Grails handles the data model.

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

Let's create a `Quote` domain class:

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

You'll see that Grails responds by creating a fresh domain class. Here's a matching unit test to get you started:

```
| Created file grails-app/domain/qotd/Quote.groovy
| Created file test/unit/qotd/QuoteSpec.groovy
```

In your Grails application, domain classes always appear under the /grails-app/domain. Look at the skeleton class Grails created in /grails-app/domain/qotd/Quote.groovy:

```
package qotd

class Quote {

    static constraints = {
    }
}
```

That's uninspiring as it appears now. You'll need fields in your data model to hold the various elements for each quote. Let's beef up your class to hold the content of the quote, the name of the author, and the date the entry was added:

```
package qotd

class Quote {
    String content
    String author
    Date created = new Date()
    static constraints = {
    }

}
```

Now that you've got your data model, you need to create your database schema, right? Wrong. Grails does all that hard work for you behind the scenes. Based on the definitions of the types in the previous code sample, and by applying simple conventions, Grails creates a quote table, with `varchar` fields for the strings, and `Date` fields for the date. The next time you run `grails run-app`, your 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 to set up a more persistent database.

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. The following code shows an extract from the standard DataSource.groovy file, which shows the default data source.

```
...
environments {
    development {
        dataSource {
            dbCreate = "create-drop"               #A
            url = " jdbc:h2:mem:devDb;MVCC=TRUE;LOCK_TIMEOUT=10000"      #B
        }
    }
    ...
}
```

#A Recreates database on every run
#B Specifies an in-memory database

You have two issues here. The `dbCreate` strategy tells Grails to drop and recreate your database on each run. This is probably not what you want, so let's change that to `update.` This change lets Grails know to leave your 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 H2[1] in-memory database. That's fine for test scripts, but not for product development. Let's change it to a file-based version of H2 so that you have real persistence.

The updated code is shown here:

```
...
environments {
    development {
        dataSource {
            dbCreate = "update"          #A
            url = "jdbc:h2:devDb;MVCC=TRUE;LOCK_TIMEOUT=10000"          #B
        }
    }
    ...
}
```

#A Preserves tables between runs
#B Specifies file-based database

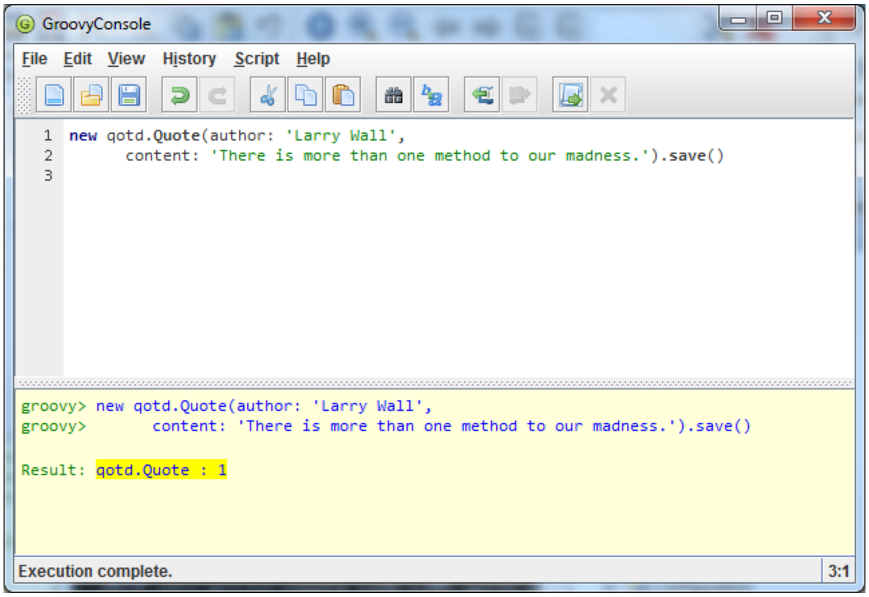
Now that you have a database that persists your data, let's populate it with sample data.

1.4.2 Exploring database operations

You haven't done any work on your user interface yet, but it would be great to save and query entries in your quotes table. To do this for now you'll use the Grails console—a small GUI application that starts your application outside a web server and gives 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 you issue this command, your QOTD Grails application is bootstrapped, and the console GUI appears, waiting for you to enter code. Figure 1.9 shows the process of saving a new quote to the database via the console.

[1] www.h2database.com



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

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

Result: qotd.Quote : 1
```

Execution complete. 3:1

Figure 1.9 The Grails console lets you run commands from a GUI.

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

```
new qotd.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: qotd.Quote : 1
```

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

```
new qotd.Quote(author: 'Chuck Norris Facts', [CA}
content: 'Chuck Norris always uses his own design patterns, [CA}
and his favorite is the Roundhouse Kick').save()

new qotd.Quote(author: 'Eric Raymond', [CA]
content: 'Being a social outcast helps you stay concentrated [CA]
on the really important things, like thinking and hacking.').save()
```

Let's use another dynamic method, `count()`, to make sure that your data saved to the database correctly (we show the script output after >>>):

```
println qotd.Quote.count()
```

```
>>> 3
```

Looks good so far. It's getting tedious typing in that `qotd` package name before each command, so let's put an import into your script to cut down on the boilerplate and get on with business:

```
import qotd.*
println Quote.count()
>>> 3
```

Much clearer. Next it's time to roll up your sleeves and query your `Quote` database. To simplify database searches, Grails introduces special query methods on your domain class called *dynamic finders*. These special methods use the names of fields in your domain model to make querying as simple as this:

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

Now that you know how to save and query, it's time to get your web application running. Exit the Grails console, and you'll learn how to get those quotes onto the web.

1.5 Adding UI actions

Let's get something on the web. To begin, you'll need an action on your `QuoteController` to return a random quote from our database. You'll work out the random selection later—for now, let's cut corners and fudge your sample data:

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

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

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

You've got a nicer data model, but nothing else is new. This is a good time to refresh your browser and see your static quote passing through to the view. Give it a try to convince yourself it's working.

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

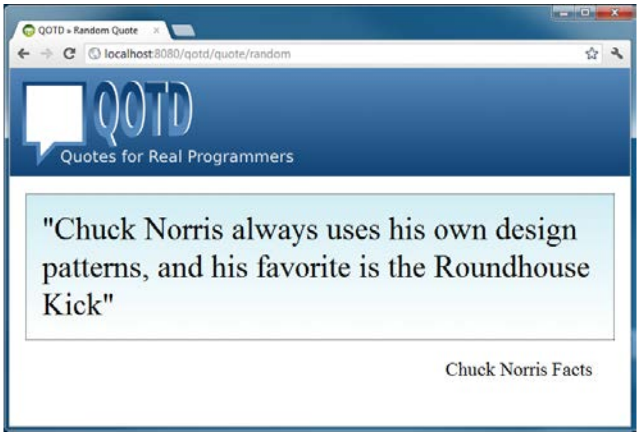
Listing 1.3 A database-driven random

```
def random() {
    def allQuotes = Quote.list()            #1
    def randomQuote
    if (allQuotes.size() > 0) {
        def randomIdx = new Random().nextInt(allQuotes.size())         #2
        randomQuote = allQuotes[randomIdx]
    } else {
        randomQuote = new Quote(author: "Anonymous",                     #3
            content: "Real Programmers Don't eat Quiche")
    }
    [ quote : randomQuote]            #4
}
```

#1 Obtains list of quotes
#2 Selects random quote
#3 Generates default quote
#4 Passes quote to the view

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

Now that you've got your random feature implemented, let's head back to http://localhost:8080/qotd/quote/random to see it in action. Figure 1.10 shows your random feature in action.



"Chuck Norris always uses his own design patterns, and his favorite is the Roundhouse Kick"

Chuck Norris Facts

Figure 1.10 Your random quote feature in action

1.5.1 Scaffolding: Adding rocket fuel

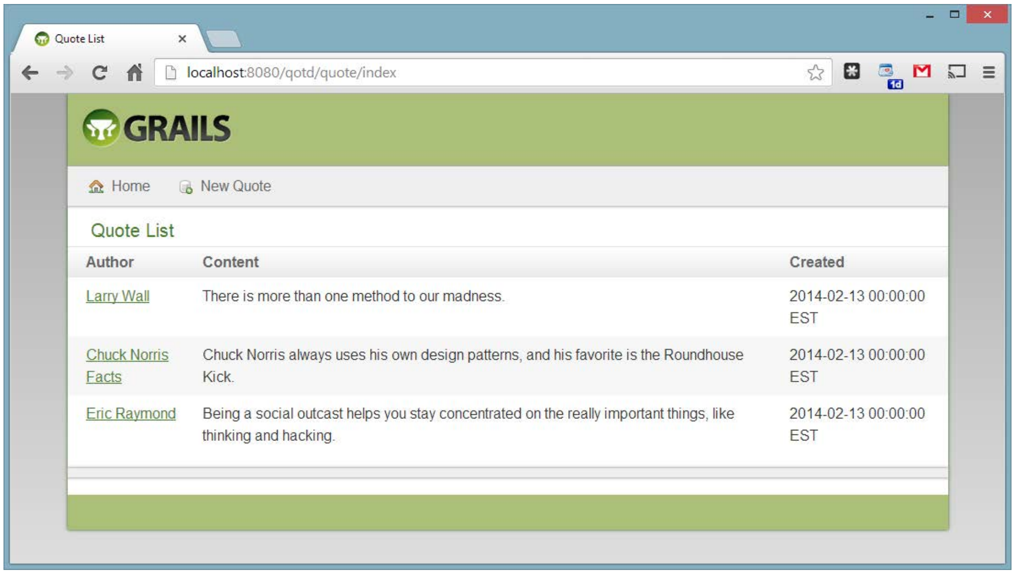
You've done all the hard work of creating your data model. Now you need to enhance your controller to handle all the CRUD actions to let users put their own quotes in the database.

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

How do you scaffold your screens for adding and updating quote-related data? It's a one-liner for the `QuoteController`, as shown in following code.

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

That's it. When Grails sees a controller marked as `scaffold = true`, it goes off and creates controller actions and GSP views on the fly. If you'd like to see it in action, head to http://localhost:8080/qotd/quote/index and you'll find something like the edit page shown in figure 1.11. (Note that this used to be called in the `list()` action if you come across code written in Grails 2.2 and earlier).

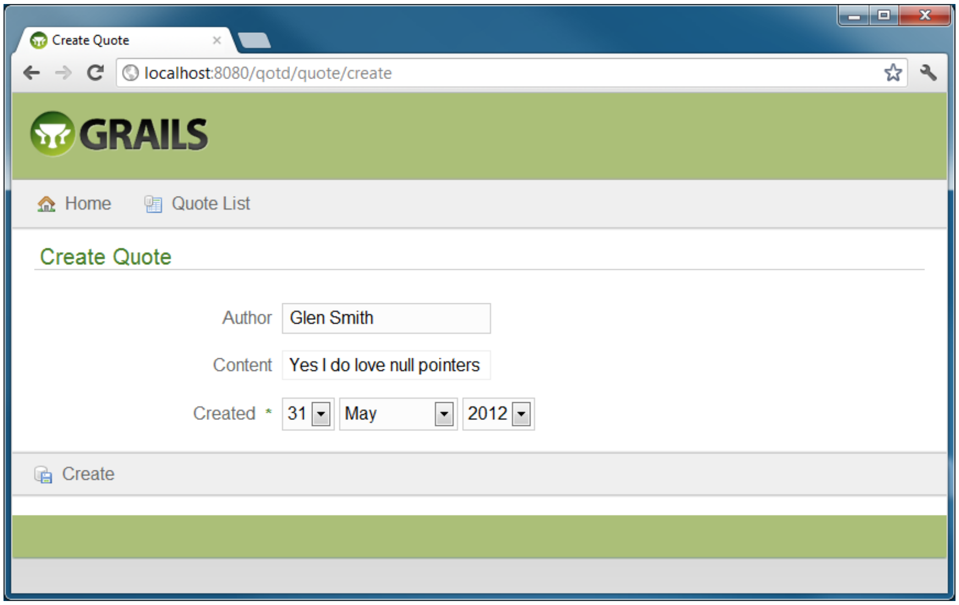


Quote List

Author	Content	Created
Larry Wall	There is more than one method to our madness.	2014-02-13 00:00:00 EST
Chuck Norris Facts	Chuck Norris always uses his own design patterns, and his favorite is the Roundhouse Kick.	2014-02-13 00:00:00 EST
Eric Raymond	Being a social outcast helps you stay concentrated on the really important things, like thinking and hacking.	2014-02-13 00:00:00 EST

Figure 1.11 The `index()` scaffold in action

Click the New Quote button, and you're up and running. You can add your new quote as shown in figure 1.12.



Create Quote

Author Glen Smith

Content Yes I do love null pointers

Created * 31 May 2012

Figure 1.12 Adding a quote has never been easier.

See how much power you get for free? The generated scaffolds aren’t 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 multiple CRUD screens).

1.5.2 Surviving the worst-case scenario

Your model looks good and your scaffolds are great, but you’re still missing pieces to make things more robust. You don’t want users putting dodgy stuff in your database, so let’s explore some validation.

Validation is declared in your `Quote` object, so you need to populate the `constraints` closure with all the rules you’d like to apply. For starters, make sure that users always provide a value for the author and content fields, as shown in the following code:

```
package qotd

class Quote {
    String content
    String author
    Date created = new Date()

    static constraints = {                      #A
        author(blank:false)                 #A
        content(maxSize:1000, blank:false)        #A
    }                 #A
}
```
#A Enforces data validation

These constraints tell Grails that neither `author` nor `content` can be blank (neither `null` nor `0` length). If you don't specify a size for `String` fields, they'll be defined `VARCHAR(255)` in your database. That's probably fine for `author` fields, but your content may expand on that. That's why you 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>` elements rather than `<input>` fields. Figure 1.13 shows how error messages display when constraints are violated.

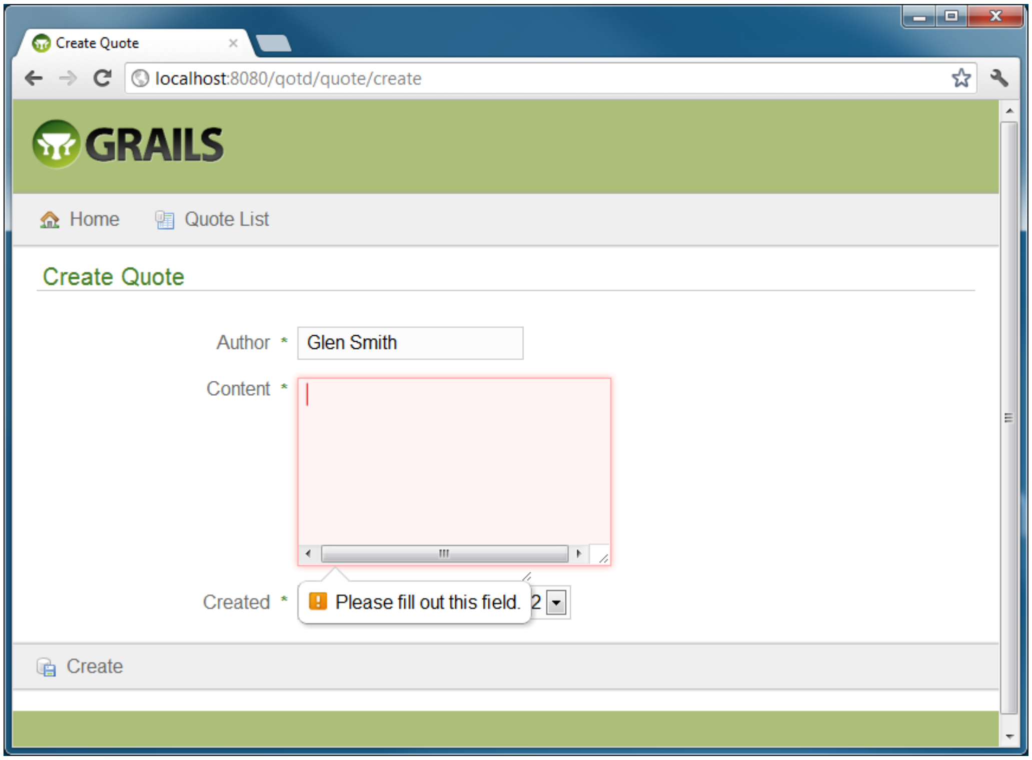


Figure 1.13 When constraints are violated, error messages appear in red.

1.6 Improving the architecture

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

Let's create your service and learn by doing:

```
grails create-service quote
```

which echoes back the familiar Grails artifact creation messages to let you know it's done:

```
| Created file grails-app/services/qotd/QuoteService.groovy
| Created file test/unit/qotd/QuoteServiceSpec.groovy
```

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

```
package qotd

import grails.transaction.Transactional

@Transactional
class QuoteService {
    def serviceMethod() {
    }
}
```

With your service created, let's rehome your random quote business logic into its own service method, as shown in the following listing.

Listing 1.4 Beefing up service

```
package qotd

import grails.transaction.Transactional

@Transactional
class QuoteService {

    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 that your service is implemented, how do you use it in your controller? Again, conventions come into play. You'll add a new field to your controller called `quoteService`, and Grails will inject the service into the controller:

```
class QuoteController {
    static scaffold = true
    def quoteService
    // other code omitted
    def random = {
        def randomQuote = quoteService.getRandomQuote()
        [ quote : randomQuote ]
    }
}
```

Doesn't that feel much tidier? Your `QuoteService` looks after all the business logic related to quotes, and your `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'll recognize this pattern of application design as dependency injection (DI). Grails takes DI to a new level by using the convention of variable names to determine what gets injected. But you have yet to write a test for your business logic, so now's the time to explore Grails' support for testing.

Services pre-Grails 2.3

The @Transactional annotation is new to Grails 2.3. In earlier versions of Grails, services were transactional by default. Don't try to add the annotation to your services if you are using one of those earlier versions.

1.6.1 Your 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 you created your `QuoteService`, Grails automatically created a skeleton unit-test case in /test/unit/qotd/QuoteServiceSpec.groovy to encourage you to test.

Grails tests are written in a testing framework called Spock. You'll learn the basics of Spock testing in Chapter 2 where we give you a proper introduction to the framework. For now, just consider Spock a "JUnit-like" testing framework where tests follow a more formal given/when/then structure.

Tests pre-Grails 2.3

Versions of Grails prior to 2.3 created standard JUnit tests rather than Spock ones. Chapter 2 shows you how to use Spock with those earlier versions.

Let's look at the skeleton test case that Grails generated.

```
package qotd

import grails.test.mixin.TestFor
import spock.lang.Specification

/**
 * See the API for {@link grails.test.mixin.services.ServiceUnitTestMixin}
for usage instructions
 */
@TestFor(QuoteService)
class QuoteServiceSpec extends Specification {

    def setup() {
    }

    def cleanup() {
    }

    void "test something"() {
    }
}
```

It's not much, but it's enough to get started. The same convention over configuration rules apply to tests, so let's beef up your `QuoteServiceSpec` case to inject the service that's under test as shown in the following listing.

Listing 1.5 Adding real tests

```
}
package qotd

import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(QuoteService)    #1
class QuoteServiceSpec extends Specification {

    void "static quote service always returns quiche quote"() {

        when:
        Quote staticQuote = service.getStaticQuote()     #2

        then:
        staticQuote.author == "Anonymous"
        staticQuote.content == "Real Programmers Don't eat quiche"
        }
    }
```

#1 Type of service to inject
#2 Injects service dynamically at runtime

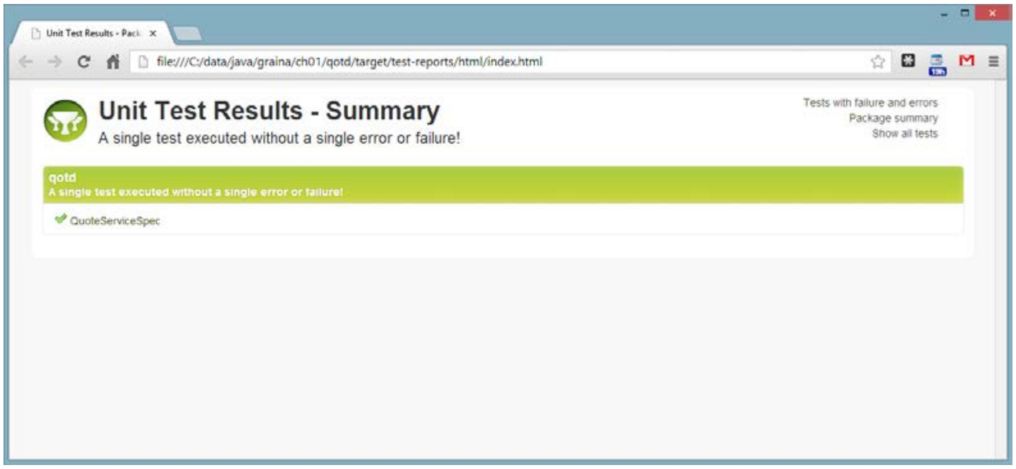
Not much can go wrong with the `getStaticQuote()` routine, but let's give it a workout for completeness.

The Grails testing framework makes heavy use of Groovy Mixins at runtime (you'll learn about these in chapter 2) to decorate your test class with magic handles. In this example we've declared this test a `@TestFor(QuoteService)` (#1). This annotation tells Grails to automatically inject a service object to the test scope that points to an instance of a real `QuoteService` object (#2).

To run your tests, execute `grails test-app QuoteServiceSpec`. If you omit the test name, it runs all the tests, but in case you're after your newly minted test case. You should see something like the following results:

```
| Tests PASSED - view reports in target\test-reports
```

This code shows that your tests run fine. Grails also generates an HTML version of your test results, which you can view by opening /target/test-reports/html/index.html in a web browser. From there you can visually browse the entire project's test results and drill down to individual tests to see what failed and why, as shown in figure 1.14.


Unit Test Results - Pack
file:///C:/data/java/graina/ch01/qotd/target/test-reports/html/index.html
Tests with failure and errors
Package summary
Show all tests

Unit Test Results - Summary

A single test executed without a single error or failure!

qotd

A single test executed without a single error or failure!

✓ QuoteServiceSpec

Figure 1.14 HTML reports from the unit test run

You'll learn how to amp up your test coverage in chapter 9, but for now you have a test up and running, and you know how to view the output.

1.6.2 Going Web 2.0: Ajaxing 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 don’t know Ajax, it’s a way of updating portions of a web page using JavaScript. Use Ajax to make your web application more responsive by updating the quote without having to reload the masthead banners and other page content. It also gives you a chance to look at Grails tag libraries.

Let’s Ajaxify your random.gsp view:

- Add the Ajax library to the `<head>` element.

  You’ll use jQuery, but Grails also lets you use Yahoo! Interface Library (YUI), Dojo, or others:

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

- In the page body of random.gsp, add a menu section that allows the user to display a new quote or navigate to the admin screens.

  You’ll use Grails’ taglibs to create both your Ajax link for refreshing quotes and your standard link for the admin interface. The following code shows your new menu HTML. Add this snippet before the `<div>` tag that hosts the body of the page:

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

You saw these tag library calls in section 1.3.3, where you used them to generate a standardized layout for your application. In this example, you introduce a `g:remoteLink`, which is Grails’ name for an Ajax hyperlink, and `g:link`, which is the tag for generating a standard hyperlink.

When you click the Next Quote link, Grails calls the `ajaxRandom` action on the controller that sent it here—in this case, the `QuoteController`—and places the returned HTML inside the div that has an ID of `quote`. But you haven’t written your `ajaxRandom` action, so let’s get to work. The following code shows the updated fragment of `QuoteController.groovy` with the new action:

```
def ajaxRandom() {
        def randomQuote = quoteService.getRandomQuote()
        render {
            q(randomQuote.content)
            p(randomQuote.author)
        }
    }
```

You've already done the heavy lifting in your quote service, so you can reuse that here. Because you don't want your Grails template to decorate your output, you're going to write your response directly to the browser (we'll talk about more elegant ways of doing this in later chapters).

We take advantage of Grails' HTML Builder to generate an HTML fragment on the fly. To satisfy your curiosity about the markup this code generates, go to http://localhost:8080/qotd/quote/ajaxRandom and see the generated HTML, which should look like this:

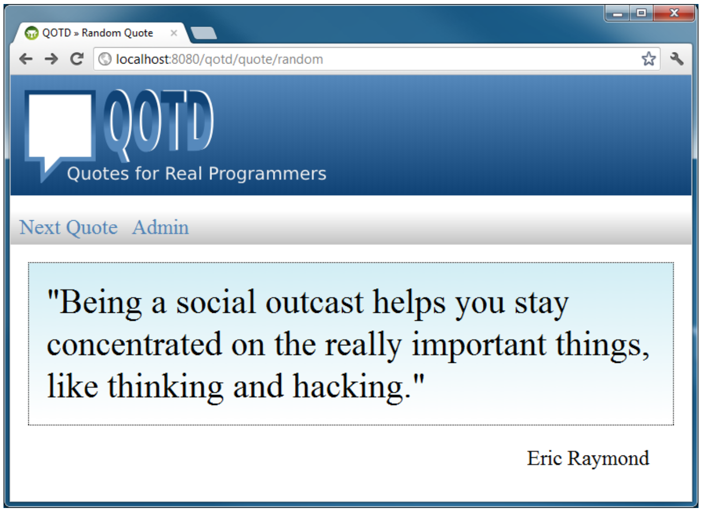
```
<q>Chuck Norris always uses his own design patterns, and his favorite is the
Roundhouse Kick</q><p>Chuck Norris Facts</p>
```

Whoa, there! What's with the embedded HTML?

In the previous sample, your render method call takes advantage of a Grails builder—a dynamic way of constructing objects of various sorts including XML, HTML, and JSON (more on these in chapter 2).

Grails also offers several other methods to achieve the same result here, including partial templates, which provide a more elegant and externalized way of achieving reusable HTML fragments. We'll talk more about this approach in chapter 8 when we discuss fragment layouts in detail.

Let's take your new Ajax app for a spin, as shown in figure 1.15.



"Being a social outcast helps you stay concentrated on the really important things, like thinking and hacking."

Eric Raymond

Figure 1.15 Your Ajax view in action

To convince yourself that all the Ajax snazziness is in play, click the Next Quote menu item several times. Did you notice 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 you'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 on the cloud. For that, you'll need a WAR file, and Grails makes its creation 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 target\qotd-0.1.war
```

Now you're ready to deploy.

1.6.4 And 80 lines of code later

You've learned about Grails. And you've created plenty of code, too. But don't take my word for it; let's have Grails crunch the numbers with a `grails stats` command. Table 1.1 shows the `grails stats` command in action.

Listing 1.6 Crunching numbers: the `grails stats` command in action

```
grails stats

    +----------------------+-------+-------+
    | Name                 | Files |  LOC  |
    +----------------------+-------+-------+
    | Controllers          |     1 |    20 |
    | Domain Classes       |     1 |    10 |
    | Services             |     1 |    20 |
    | Unit Tests           |     3 |    39 |
    +----------------------+-------+-------+
    | Totals               |     6 |    89 |
    +----------------------+-------+-------+
```

Only 89 lines of code (LOC)! Not too shabby for an Ajax-powered, user-editable, random quote web application with unit tests. If you removed the empty skeleton test cases that Grails created for your `Domain` and `Controller` classes, you could probably trim it down to 63.

This Grails introduction has given you a taste of models, views, controllers, services, taglibs, layouts, and unit tests. And you've got more to explore. But before you go further, let's explore Groovy.

1.7 Summary and best practices

Congratulations, you've written and deployed your Grails app, and now you have a feel for working from scratch to completed project. The productivity rush can be 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 fun along the way.
- *Noise reduction fosters maintenance and increases velocity.* By embracing convention over configuration, Grails eliminates 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 skeleton boilerplate code to get you up and running—another way Grails saves time.
- *Testing is inherent.* Grails makes writing test cases easy. It even creates skeleton 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.

We'll spend the rest of the book taking you through 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.