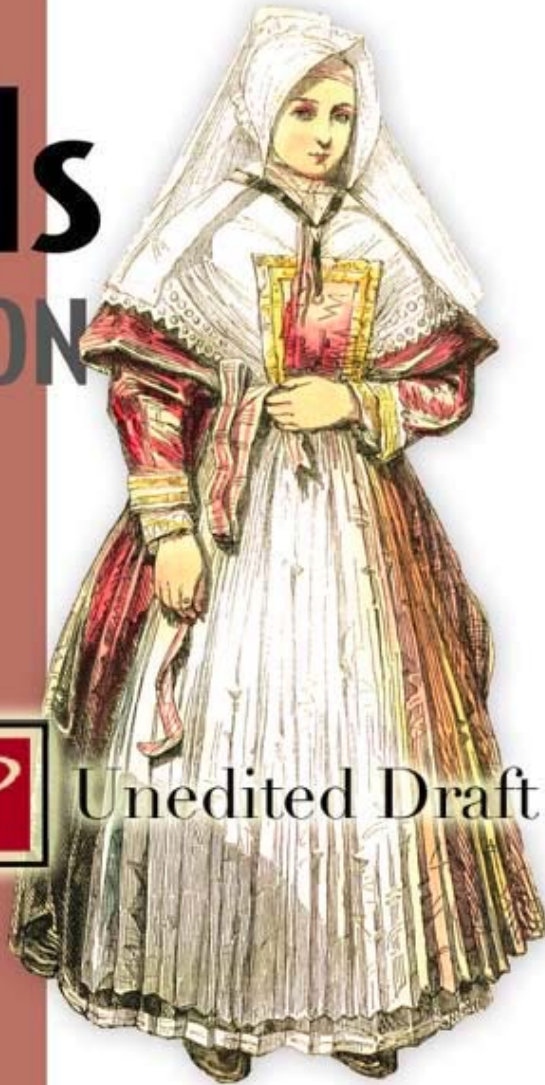


Grails IN ACTION

Glen Smith
Peter Ledbrook

MEAP

Unedited Draft



MANNING



**MEAP Edition
Manning Early Access Program**

Copyright 2008 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=493>

Table of Contents

Part 1: Introduction

Chapter 1: Grails in a Hurry

Chapter 2: Groovy Essentials

Part 2: Fundamentals

Chapter 3: Life after SQL, this ain't your Grandma's OUTER JOIN

Chapter 4: Controllers

Chapter 5: Developing Tasty Views, Forms and Layouts

Chapter 6: Building reliable applications

Part 3: Building more features into your applications

Chapter 7: Using Plugins: Adding Web 2.0 in 60 minutes

Chapter 8: Wizards and Workflow with Webflow

Chapter 9: Don't let strangers in – security

Chapter 10: Remote access

Chapter 11: Messaging for Fun and Profit

Part 4: What you need to know for real work

Chapter 12: Advanced GORM Kungfu

Chapter 13: In the Belly of the Beast – Spring and Transactions

Chapter 14: Beyond compile, test, and run

Chapter 15: Plugin Development

1

Grails in a Hurry...

One of the big wins of using an opinionated, agile web framework like Grails is providing a substantial productivity boost – you can develop webapps in record time. So through the course of this chapter we'll talk a gentle walk through developing a simple Grails application – giving you a “sip from the firehose” – a taste of the kind of productivity you can expect when moving to Grails.

Most programmers I know are the impatient type, so this chapter will be a pretty fast-paced one. So we'll take 30 minutes, and develop a data-driven, Ajax powered, Unit tested, Web 2.0 monster.... well maybe not a *monster*... but a deployable public website none-the-less. Along the way you'll get a taste of all the core parts of a Grails applications: Models, Views, Controllers, Taglibs and Services. Buckle up, it's time to hack.

1.1 Getting Setup

The journey of a thousand miles begins with the first download... or something like that. In order to get Grails up and running, you'll need to have a JDK installed (v1.4 or later – try a `javac -version` from your command prompt to confirm). Once you're happy that your JDK is all good, download the latest Grails distro from <http://www.grails.org> and unzip it to your favourite installation area. You'll then need to set an environment variable called `GRAILS_HOME` which points to your Grails installation directory, and finally, add `GRAILS_HOME/bin` to your path.

On OSX and Linux, this is normally done by editing your `~/.profile` script to contain something like:

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

On Windows, you'll need to go into XXXXX

((screen shot of Windows environment settings))

((checklist: jdk installed (`javac -version`), `GRAILS_HOME` setup, `GRAILS_HOME/bin` in path))

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

```
grails version

Welcome to Grails 1.0.3 - 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.

Lastly, when you develop more sophisticated Grails applications, you'll probably want to take advantage of some of the fantastic Grails IDE support out there. A full coverage of the current crop of IDEs is found in section XXX. We won't be developing too much code in this chapter, so a basic text editor will be plenty. Fire up fave editor, and let's talk about our sample application.

1.1.1 Our Sample: A Web2.0 QOTD

If we're going to go to the trouble of writing a small application, we might as well have some fun. Our little sample is a quote-of-the-day (QOTD) application where we'll capture famous programmer quotes from development Rock Stars throughout time. We'll want to have a nice short URL for our application, so let's make “qotd” as our app's little working title.

So this is the moment you've been waiting for right? It's time to get started building our groundbreaking, world-changing Web 2.0 programmer quotation app. And all Grails projects begin the same way.... First, find a directory to work in. Done? Great. Now create the application:

```
grails create-app qotd
```

Well done. You've created your first Grails application. You'll see Grails create a subdirectory called “qotd” to hold our application files. You'll want to change to that directory now, and we'll camp out there for the rest of the chapter. Since we've done all the hard work of building the app, it'd be a shame to not enjoy the fruit of our labour. Let's give it a run:

```
grails run-app
```

Grails ships with a copy of Jetty (a tiny embedded Java application server) which Grails will use to host your application during the development and testing lifecycle. When you issue the “grails run-app” command, Grails will compile and start your web application. When everything is ready to go, you'll see a message on the console saying:

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

Which means it's time to fire up your favourite browser and take your fresh application for a spin:

```
http://localhost:8080/qotd/
```



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:

Figure 1.1 Our first app is up and running

1.2 Beefing up our Application

Well we have our application built and deployed, but we're a little short on an engaging user experience. Before we go too much further, it's probably a good time to learn a little of how Grails handles interaction with the user – and that's via a Controller.

Controllers are at the heart of every Grails application. They look after taking input from your user's web browser, interacting with your business logic and data model, and then routing the user to the correct page to display. Without Controllers, your webapp would just be a bunch of static pages.

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

```
grails create-controller quote
```

Grails will create our skeleton controller in `/grails-app/controller/QuoteController.groovy`. You'll notice that Grails sorted out the capitalisation for you. It's the small things, right? Let's take a look at the basic skeleton:

```
class QuoteController {
    def index = { }
}
```

Not so exciting. The index entry you see above is called a Grails "action" – which we'll return to the significance of in a moment. For now, let's add a home action which sends some text back to the browser:

```
class QuoteController {
```

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=493>

```

    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.

So how do we invoke our action in a browser? If this is a Java web application, the url to get to it must be declared in some configuration file, right? Hmm. Actually. No. This is where I need to introduce you to the idea of "convention over configuration".

Ruby on Rails introduced the idea that tons of XML configuration (or any sort really) can be avoided if the framework makes some opinionated choices for you about how things will fit together. Grails embraces the same philosophy. Since our controller is called `QuoteController`, Grails will expose its actions over the url `/qotd/quote/youraction`. So in the case of our hello action, we'll need to head on over to:

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

And we're in business. Without a single line of XML!

Real Programmers do not eat Quiche

Figure 1.2 Adding our first bit of functionality

Figure 1.3 Hedgehog diagram of URL to grails bits and pieces

If you were wondering about that `index()` routine, that's the method that gets called when the user omits the action name. If we decide that all references to `/qotd/quote/` should end up `/qotd/quote/home` then we need to tell Grails about that:

```

class QuoteController {
    def index = {
        redirect(action: home)
    }

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

```

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.

1.2.1 Writing stuff out: the view

Embedding html inside your code is always a bad idea. It's difficult to read and maintain, and it eliminates any chance for your graphic designer to work their magic without getting access to your source code. The answer is to move your display logic out to a separate file for the html display – which is known as the “view” - and Grails makes it a snack.

If you've done any Java Web applications work, you'll be familiar with the humble JSP (Java Server Page). JSPs look after displaying html to the user of your web application. Grails applications, conversely, make use of a GSP (or Groovy Server Page). The concepts are quite similar, and you'll find the shift is fairly small.

We've already discussed the “convention over configuration” model, 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, let's implement our random action, then we'll worry about the view:

```
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 your controller action can pass information on to your view. If you're an old-school servlet programmer, you might think of it as “request”-scoped data. The [] operator in Groovy is a Map, so we're passing a series of key/value pairs through to our view.

So... where does our view actually fit into this? And where will we put our GSP file so that Grails knows where to find it? Following on from our convention-over-configuration bent, we'll follow the naming conventions of the controller, coupled with the name of our action and will end up in placing our gsp in /grails-app/views/quote/random.gsp. If we follow that pattern, there's no configuration required.

So let's create a GSP and see how we can reference our map data:

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

<body>

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

</body>
</html>
```

The use of the \${content} and \${author} format is known as the GSP “Expression Language” and if you've ever done any work with JSPs, it probably will be old news to you. Let's fire up the browser and give it a whirl, shall we? Put url here

“Real Programmers don't eat much quiche”

Anonymous

Figure 1.4 Our first view in action

1.2.2 Adding some style with Grails Layouts

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

I know what you're thinking: it's time for some css action. But let's plan ahead a little. If I markup random.gsp with my funky css goodness, I'm going to have to add those links to the header of every page in my app. There must be a better way... and the answer is Grails Layouts.

Layouts give you a way of specifying “template” layouts for certain parts of your application. For example, we might want all of the quote pages (random, by author, by date) quotes to be styled with a common masthead, and navigation links. Basically we only want the body content of the page to change, and keep the rest the same. First let's markup our target page with some ids that we can use for our CSS:

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

<body>

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

</body>
</html>
```

Now, how to apply that skinning? Like everything else in Grails, Layouts follow a convention-over-configuration style. So for all our QuoteController actions to share the same layout, we'll create a file in /grails-app/layouts called quote.gsp. There are no Grails shortcuts for layout creation, so we've got to roll this one by hand. Here's our first attempt:

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=493>

```

<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">
      
    </div>
    <g:layoutBody />
  </body>
</html>

```

Wow... that's a lot of angle brackets. Let's break it down. The key thing to remember is that we're in a template page, so contents of our target (`random.gsp`) page will be merged with this template before we send any content back to the browser. That means that we need a way of accessing elements of the target page (eg to merge the title of the target page with the template). Enter the Grails template taglibs.

We've made use of a few Grails taglibs here which you haven't see before, so let's take moment to have a look. In the title block of the page we include our QOTD title, then follow it with some chevrons (`>>`) then add the title of the target page itself.

After the rest of the head tags, we do a `layoutHead` call to merge in the contents of the HEAD section of any target page. This can be important for Search Engine Optimisation (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. First we output our common masthead div to get our Web2.0 gradient and cute icon goodness, then do a call to `<g:layoutBody>` to render the BODY section of the target page.

Let's do a refresh to see how we're trucking:



“Real Programmers don't eat much quiche”

Anonymous

Figure 1.5 That's better. Now with some funky CSS skinning

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=493>

Looking good. Notice how we've made no changes to our relatively bland `random.gsp`? Keeping your view pages free of this sort of cosmetic markup means that your maintenance overhead is reduced significantly. Need to change your masthead, add some more javascript includes, or 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 storing stuff in the database. Once that's under our belt, we can circle back and implement a real random action.

1.3 Getting some user input, Creating your Domain model

Our app is underway and we can deploy it to our testing web container. We've come a long way in a few short sections. But let's not overstate our progress – Google is not 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. But in order to store those quotations, we're going to need to learn some essentials of how Grails handles your Data Model.

Grails uses the term "Domain Class" to describe those objects in your system that are capable of being persisted to the database. In our qotd system we're probably 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 our Quote domain class:

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

In your Grails application, domain classes always end up under `/grails-app/domain`. Let's have a look at the skeleton class Grails has created for us in `/grails-app/domain/Quote.groovy`:

```
class Quote {
}
```

Hmm. That's pretty uninspiring. We're going to need some fields in our data model to hold the various elements on each quote. Let's beef up our class to hold the content of the quote, the name of the author, and the date we added this entry.

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

So we've got our Data Model, now we need to go off and create our database schema, right? Wrong. Grails does all that hard work for you under the covers. Based on your definitions of the types above, and applying some simple conventions, Grails will create a "Quote" table, with varchar fields for the Strings, and Date fields for the Date. So next time we do a "grails run-app" our datamodel will be created on the fly. But how will it know which database to create the tables in? It's time to configure up a DataSource.

1.3.1 Configuring your Data Source

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

In your `/grails-app/conf/` directory, you'll find a file named `DataSource.groovy`. This is where you define the datasource (database) that your application will use. The file allows you to define different databases for your development, test and production environments. When you do "grails run-app" to run the local web server, it uses your development one. So what is the default dev datasource?

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

Yes. Well. We have two issues here. The first is the `dbCreate` strategy. This 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 and remove columns if it needs to).

The second issue relates the url. It's using an HSQL in-memory database. That's fine for your test scripts but not so good for your product development. Let's change it to a "file" based version of HSQL so we have some real persistence going on.

Our updated file will read something like this:

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

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

1.3.2 Exploring Database Operations

Now we haven't done any work on our user interface yet, but it would be great to be able to explore saving and querying entries to our quotes table. What's a curious Grails developer to do? Enter the Grails Console – a small GUI application which 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.

```
grails console
```

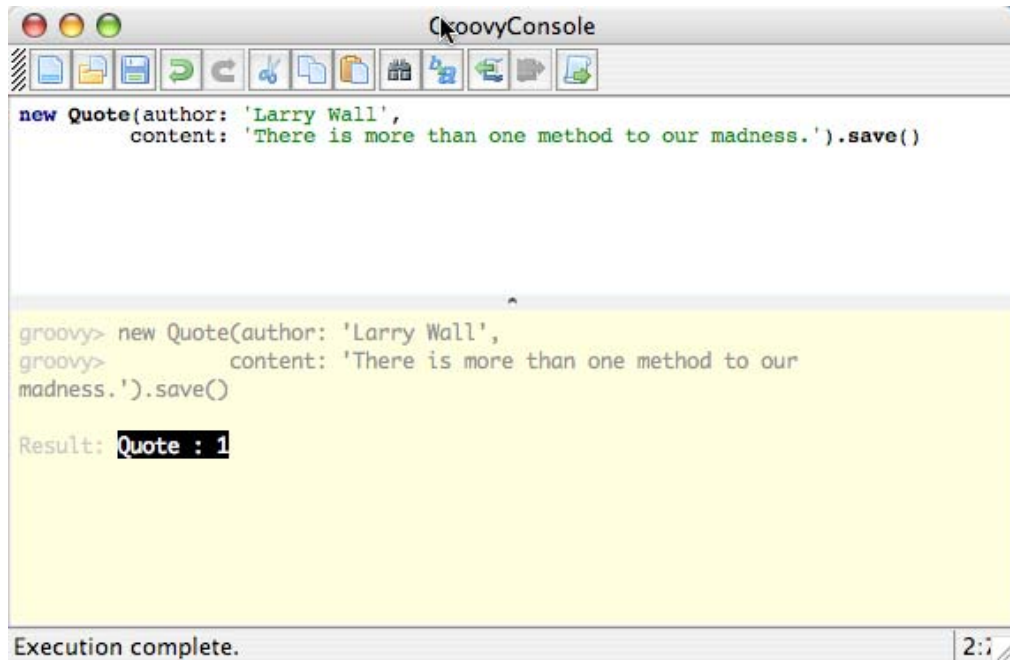


Figure 1.6 The Grails console let's your run commands from a GUI

Now that we have a console, and our Grails application is bootstrapped, it would be nice to create and save some of those Quote objects. Type the following into the console window, and click the Run button (far right of the toolbar):

```
new Quote(author: 'Larry Wall',
          content: 'There is more than one method to our madness.').save()
And the bottom half of the console will let you know we're on track:
```

```
Result: Quote : 1
```

Whoa there! Where did that save() routine come from? I didn't see it in our class definition? You're right. Grails automatically endows domain 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 to make sure that our data is getting 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 make database searches super-simple, Grails introduces special query methods on your domain class called "dynamic finders". These special methods utilise the names of fields in your domain model to make querying as simple as:

```
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 probably time to start getting our web application up and running, so exit out of the Grails console, and let's learn a little about getting those Quotes onto the web...

1.4 Adding some UI Action

Enough Data Modelling, let's get something on the web! First on the list is an action on our QuoteController to return a random quote from our database. Just to get used to how things hang together, let's cut some corners and fudge our sample data:

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

And we'll need to update our /grails-app/views/quote/random.gsp to use our new Quote object, but that's just a change to the Expression Language:

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

Nothing new here, just a nicer data model. But now that we know enough GORM to be dangerous, let's rework our action to implement a real random database query:

```
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]
}
```

((annotate the example line by line))

So 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:

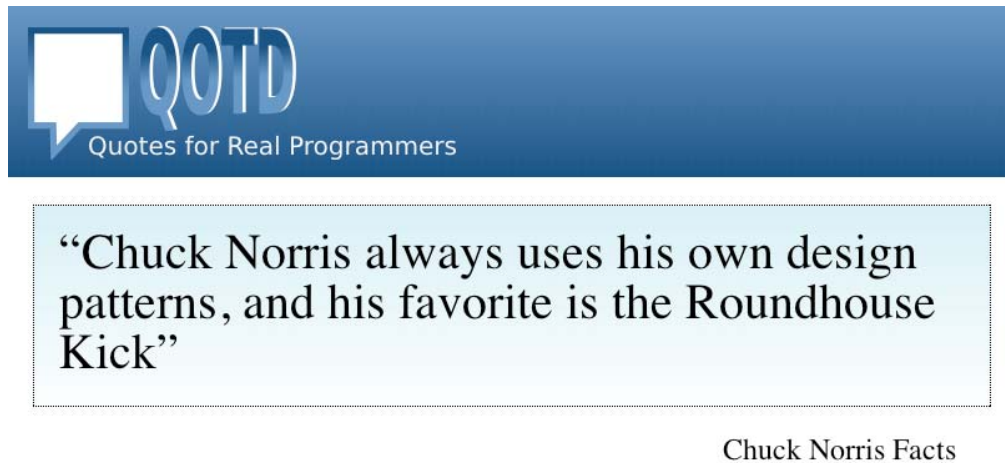


Figure 1.7 Our random quote feature in action

1.4.1 Scaffolding: Just add Rocket Fuel

So we've done all the hard work of creating our Data Model, now we'll probably need to enhance our controller to handle all the CRUD (Create/Read/Update/Delete) actions that we'll need to actually let the user put their own quotes in the database?

To do a really slick job of it, probably yes. But if we want to just 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. So... how do we scaffold up our screens for adding and updating Quote-related data? It's a one-liner for the QuoteController:

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

And 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 on over to <http://localhost:8080/qotd/quote/list> and you'll find something like this:



Home New Quote

Quote List

| Id | Author | Content | Created |
|----|-----------------------|---|----------------------------|
| 1 | Larry Wall | There is more than one method to our madness. | 2008-06-26 21:02:12.943 |
| 2 | Chuck Norris Facts | Chuck Norris always uses his own design patterns, and his favorite is the Roundhouse Kick | 2008-06-26 21:02:13.558 |
| 3 | Eric Raymond | Being a social outcast helps you stay concentrated on the really important things, like thinking and hacking. | 2008-06-26 21:02:13.561 |

Figure 1.8 The list() scaffold in action

Click on the “New Quote” button, and you’ll be up and running to add your new quote:



Home Quote List

Create Quote

Author:

Content:

Created: :

Figure 1.9 Adding a Quote has never been easier

Boy 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 just perfect for tinkering with your database during development (where you don’t want the overhead of mocking together a bunch of CRUD screens).

1.4.2 Surviving the Worst Case Scenario

Our Model is looking good, our scaffolds are all going great, but we're still missing some pieces to make things a little more robust. We don't want user putting dodgy stuff in our Database, so it's probably a good time to explore some validation.

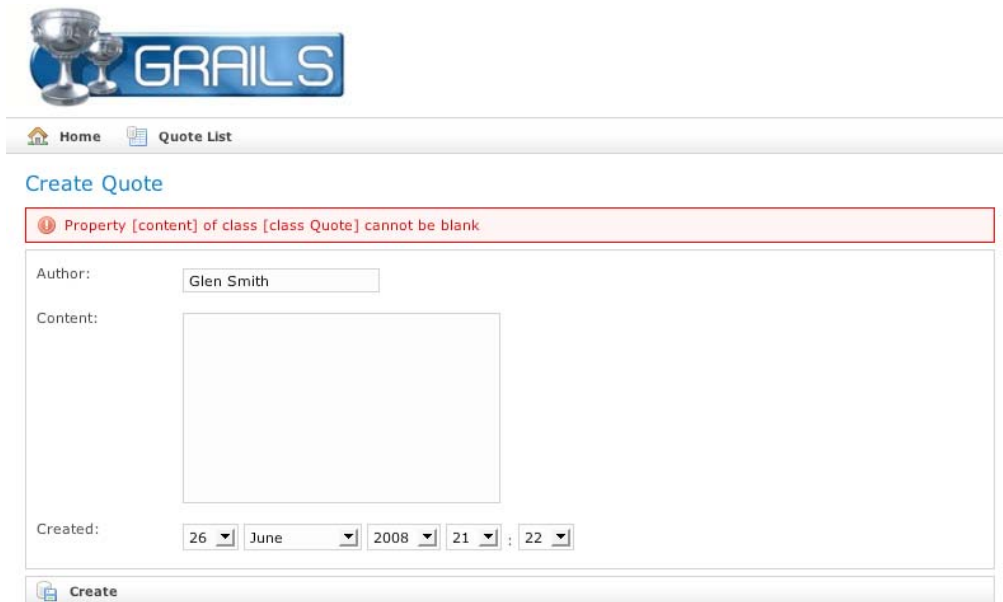
Validation is accomplished through definitions directly on our Quote object – so there's just one place that you need to look for constraints! We need to introduce a constraints closure with all the rules we'd like to apply. For starters, let's make sure that people always provided a value for our author and content fields:

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

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

This tells Grails that neither author or content can be blank (neither null or 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 just 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 affects things in the generated scaffolds. For example, the ordering of entries in the constraints closure also affect what order the fields appear in when generated pages. Fields with constrain sizes greater than xxx chars are rendered as html TEXTAREAs rather than TEXT fields.



The screenshot shows a web application interface for creating a quote. At the top, there is a navigation bar with 'Home' and 'Quote List' links. Below this is a header for 'Create Quote'. A red-bordered error message box at the top of the form area contains the text: 'Property [content] of class [class Quote] cannot be blank'. The form itself has three main sections: 'Author:' with a text input field containing 'Glen Smith'; 'Content:' with a large, empty text area; and 'Created:' with a date and time selector showing '26 June 2008 21:22'. At the bottom of the form is a 'Create' button.

Figure 1.10 When bad data happens to good apps

1.5 Improving the Architecture

Having tons of logic spread 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 is not really a concern right now. But as our quotation app grows, we'll find that things get a little more complex. We'll want to re-use 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
```

And we'll find ourselves with 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 of that later. For now let's move our random quote business logic into it's own method on the 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
    }
}
```

So now our service is implemented. How do we get to it in our controller? Again, conventions come into play. It's as easy as adding a new field to our controller called "quoteService" and Grails will inject the service into the controller ready-to-go:

```
class QuoteController {
    def scaffold = true
    def quoteService
    def random = {
        def randomQuote = quoteService.getRandomQuote()
    }
}
```

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=493>

```

        [ quote : randomQuote ]
    }
}

```

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

1.5.1 Loving the Green Bar: Your first Grails test case

Testing is certainly 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 test case to encourage us to do the right thing. Let's have a look in `/grails-app/test/integration/QuoteServiceTests.groovy`:

```

class QuoteServiceTests extends GroovyTestCase {
    void testSomething() {
    }
}

```

Well that's not a whole lot of encouragement, but it's enough to get us started. The same "convention-over-configuration" rules apply to tests, so let's beef up our `QuoteServiceTest` case to inject the service that's under test:

```

class QuoteServiceTests extends GroovyTestCase {
    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 unit tests, execute a "grails test-app" and wait for the output. You should see something like:

```

-----
Running 3 Integration Tests...
Running test QuoteControllerTests...
    testSomething...SUCCESS
Running test QuoteServiceTests...
    testStaticQuote...SUCCESS
Running test QuoteTests...
    testSomething...SUCCESS
Integration Tests Completed in 284ms
-----

```

Which shows us that our tests are running just fine. If you wondered where those other tests came from, they were the test shells that Grails created for us when we created our Model class and our Controller class.

Grails also writes out a html version of our test results, which you can find by opening `/grails-app/test/reports/html/index.html`. From there you can browse the whole project's test results visually and drill down into individual tests to see what failed and why.

The screenshot shows a web page titled "Unit Test Results" with a sidebar on the left containing navigation links for Home, Packages, and Classes. The main content area displays a summary table and a packages table.

Unit Test Results
Designed for use with [JUnit](#) and [Ant](#).

Summary

| Tests | Failures | Errors | Success rate | Time |
|-------|----------|--------|--------------|-------|
| 3 | 0 | 0 | 100.00% | 0.180 |

Note: *failures* are anticipated and checked for with assertions while *errors* are unanticipated.

Packages

| Name | Tests | Errors | Failures | Time(s) | Time Stamp | Host |
|--------|-------|--------|----------|---------|---------------------|-------------|
| <none> | 3 | 0 | 0 | 0.180 | 2008-06-26T11:30:13 | decaf.local |

Figure 1.11 Pretty reports from the Unit test run

We'll learn how to amp up our test coverage kungfu later in the book, but for now we're satisfied that we have a test up and running, and we know how to view the output.

1.6 The whole 9 yards: Going Web2.0: 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 sneaky way of updating portions of a web page using JavaScript. By using a little AJAX, we can make our little web application a lot more responsive by only updating the quote itself 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.

First up, let's ajax our random.gsp view. First we have to add the Ajax library to our `<head>` element (we'll use Prototype, but Grails lets you use YUI, Dojo, and a wealth of others). Here's the updated portion of random.gsp

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

Then in the page body of random.gsp, we'll add a menu section which allows the user to ajax a new quote, or head off to the admin screens. We'll use one of Grails taglibs to create our AJAX link, and also the standard link to the admin interface:

```
<ul id="menu">
  <li>
    <g:remoteLink action="ajaxRandom" update="quote">
```

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=493>

```

        Next Quote
    </g:remoteLink>
</li>
<li>
<g:link action="list">
    Admin
</g:link>
</li>
</ul>

```

If you've never seen a tag library before, think of them as a custom html tag that can execute a bunch of code. In this case our taglib is a "g:remoteLink" – Grails name for an Ajax hyperlink – or a "g:link" which is the normal href that you know and love.

When you click on this link, Grails will call the "ajaxRandom" routine on whatever controller sent it here – in our case the QuoteController – and will place the returned html inside the div which has an id of "quote". But we haven't written our ajaxRandom routine on that controller, so let's get to work. Let's update QuoteController.groovy with the new routine:

```

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

```

We'd already done the heavy lifting in the service, so we can re-use our hard work again 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:

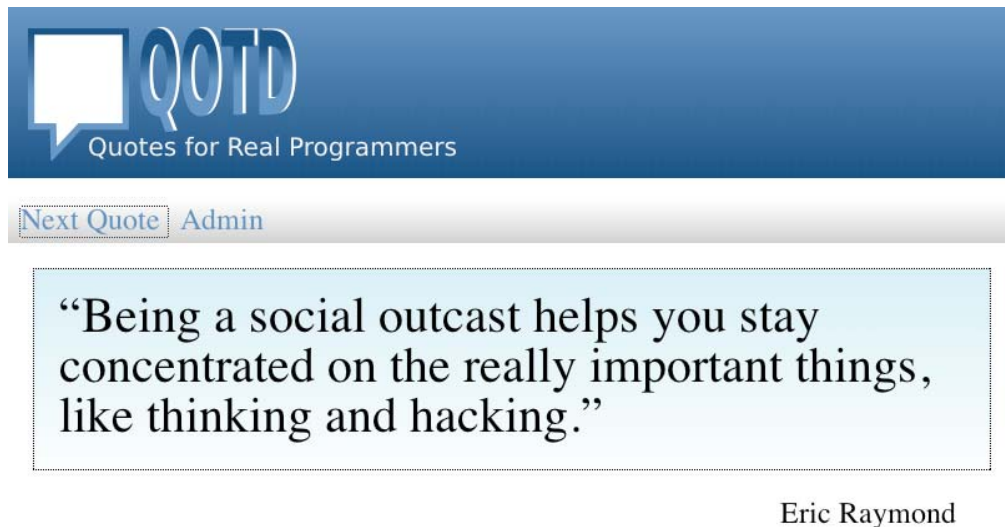


Figure 1.12 Our Ajax view in action

Just 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 Web2.0 dream.

1.6.1 Bundling The Final Product: Creating a war

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

```
grails war
```

Follow along the output and you'll see Grails bundling up all the jars it needs along with your Grails application, and dropping the bundled war in your project's root directory:

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

And you're ready to deploy.

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:

```
decaf:~/qotd glen$ grails stats
```

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

Wow. 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 is not too shabby for an Ajax-powered, user editable, random quote web application!

That was quite a "sip from a firehose" view of Grails. We've had a taste of models, views, controllers, services, taglibs, layouts and unit tests. But there is so much more to explore. But before we get too much further, it might be good to explore a little groovy...