



**Scott Davis**  
**Jason Rudolph**

# **Getting Started with Grails** **Second Edition**

**A modern web framework for the Java platform**

ENTERPRISE SOFTWARE  
DEVELOPMENT SERIES

**InfoQ**  
new

# FREE ONLINE EDITION

If you like the book, please support  
the authors and InfoQ by

**purchasing the printed book:**

**<http://www.lulu.com/content/7868425>**

**(only \$22.95)**

Brought to you  
Courtesy of



This book is distributed for free on InfoQ.com, if  
you have received this book from any other  
source then please support the authors and the  
publisher by registering on InfoQ.com.

**Visit the homepage for this book at:**

**[http://www.infoq.com/minibooks/grails-  
getting-started](http://www.infoq.com/minibooks/grails-getting-started)**

# ***Getting Started with Grails***

**Second Edition**

**Written by  
Scott Davis  
and  
Jason Rudolph**

**Free Online Version**

Support this work, buy the print copy:

[http://www.infoq.com/minibooks/grails-  
getting-started](http://www.infoq.com/minibooks/grails-getting-started)

© 2010 C4Media Inc  
All rights reserved.

C4Media, Publisher of InfoQ.com.

This book is part of the InfoQ Enterprise Software Development series of books.

For information or ordering of this or other InfoQ books, please contact [books@c4media.com](mailto:books@c4media.com).

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recoding, scanning or otherwise except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without the prior written permission of the Publisher.

Designations used by companies to distinguish their products are often claimed as trademarks. In all instances where C4Media Inc. is aware of a claim, the product names appear in initial Capital or ALL CAPITAL LETTERS. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

Managing Editor: Diana Plesa  
Cover art: Bistrian Iosip  
Composition: Accurance

Library of Congress Cataloguing-in-Publication Data:

ISBN: 978-0-557-18321-0

Printed in the United States of America

## Scott's Acknowledgements (2<sup>nd</sup> Edition)

---

Jason Rudolph did an outstanding job with the 1<sup>st</sup> edition of this book. It is quite literally how I cut my teeth with Grails. The fast pace of the book – never lingering too long on any one topic, but covering everything that you need to get a complete application up and running – was a welcome respite from the 1,000 page boat-anchors typical of technical books.

As he and I got to know each other over the years – speaking at the same conferences, raising a pint or two along the way – I would always hassle him with the same question: “So, when are you going to put out a 2<sup>nd</sup> edition of *Getting Started with Grails*? Dude, you *have* to do it...” The fact that I am the co-author of this edition speaks volumes about Jason’s business acumen – never say “No” to a man who is both eager *and* buying you beer. So thank you, Jason, for allowing me to bring your baby into the modern era of Grails.

Thanks also to Graeme Rocher, Jeff Brown, and the rest of the core Grails development team. The evolution from open source project to incorporating as G2One to being acquired by SpringSource (and then by VMWare) in such a short time must be incredibly rewarding —as if being the #1 mailing list in traffic volume at codehaus.org or achieving 90,000 downloads in a single month isn’t rewarding enough. You have managed to put together a not just a web framework, but a community. Good on ya, mates.

And speaking of mates, my buddies from the conference circuit continue to be a great source of inspiration and positive reinforcement. Neal Ford, Stuart Hallaway, Venkat Subramaniam, David Geary, Andy Glover, David Bock, Brian Goetz, David Hussman, Ted Neward, Michael Nygard, Mark Richards, Brian Sam-Bodden, Nate Shutta, Ken Sipe, and Brian Sletten – thanks for the late nights, airport camaraderie, and Mexican buffets every Sunday. *In a mad world, only the mad are sane.* (Akira Kurosawa)

Thanks to Floyd Marinescu, Alexandru Popescu, John-Paul Grabowski, and the rest of the team at InfoQ for making this 2<sup>nd</sup> edition a reality.

And, as always, thanks to my long suffering wife Kim. She's the one who says, "Isn't it about time for you to write another book?" and then regrets it for the next year as it preempts any semblance of a sane home life. She's also the one who says, "Isn't it about time for you to put away the MacBook Pro and play with Christopher and Libby?" She always seems to know the right thing to say at the right time.

## Jason's Acknowledgements

### (1<sup>st</sup> Edition)

---

I would like to thank Graeme Rocher (Grails project lead and author of *The Definitive Guide to Grails*) for reviewing this book and for enlightening me to the inner workings of Grails along the way. Your detailed review provided invaluable technical insight and excellent recommendations for incorporating “Groovier” solutions into the examples.

I would also like to thank Venkat Subramaniam (co-author of *Practices of an Agile Developer*) for reviewing this book with a keen emphasis on the learning experience and how to best introduce Grails to the reader.

I'd like to thank Steve Rollins for diligently pouring through this book to uncover any lingering issues, even though it meant a few weeks of being a nerd outside of normal working hours. Your tireless attention to detail is clearly reflected in the final product.

I would also like to thank Jared Richardson (co-author of *Ship it! A Practical Guide to Successful Software Projects*) not only for reviewing this book, but for motivating me to write it in the first place. Your encouragement throughout the process and your valuable perspective on the end result are greatly appreciated.

I would like to thank Floyd Marinescu (co-founder of InfoQ.com and author of *EJB Design Patterns*) and the entire team at InfoQ for publishing this book and for your enthusiastic support throughout the effort.

Most importantly, I'd like to thank my unflinchingly patient and encouraging wife Michelle for supporting me throughout this effort. It was only by your willingness to do far more than your fair share (of just about everything) that I was able to have time for this project. And as if it wasn't enough just to provide the support to make this book possible, your creativity, sense of style, and editorial input contributed to make this book better than it would have been on its own.





# Contents

<b>INTRODUCTION</b>	<b>1</b>
Learning by Example	1
The RaceTrack Application	2
<b>INSTALLING GRAILS</b>	<b>3</b>
Installing a JDK	3
Installing Grails	3
Installing a Database	4
<b>CREATING A GRAILS APPLICATION</b>	<b>5</b>
Creating the RaceTrack Application	5
The Grails Directory Structure	8
Domain Classes	10
Scaffolding Controllers and Views	14
<b>VALIDATION</b>	<b>21</b>
Customizing the Field Order	21
Adding Validation	25
Changing Error Messages	28
Creating Custom Validations	30
Testing Validations	34
<b>RELATIONSHIPS</b>	<b>41</b>
Creating a One-to-Many Relationship	41
Creating a Many-to-Many Relationship	47
Bootstrapping Data	51
<b>DATABASES</b>	<b>57</b>
GORM	57
DataSource.groovy	57
Switching to an External Database	59
<b>CONTROLLERS</b>	<b>65</b>
create-controller vs. generate-controller	65
Understanding URLs and Controllers	67
From Request to Controller to View	68
A Quick Look at GSPs	70
Exploring the Rest of the Controller Actions	71
Rendering Views That Don't Match Action Names	73

<b>GROOVY SERVER PAGES</b>	<b>75</b>
Understanding GSPs	75
Understanding SiteMesh	79
Understanding Partial Templates	81
Understanding Custom TagLibs	86
Customizing the Default Templates	89
 <b>SECURITY</b>	 <b>97</b>
Implementing User Authentication	97
Unit Testing Controllers	106
Creating a Password Codec	113
Creating an Authorization TagLib	117
Leveraging the beforeInterceptor	119
Leveraging Filters	122
Security Plugins	123
 <b>PLUGINS, SERVICES, AND DEPLOYMENT</b>	 <b>125</b>
Understanding Plugins	125
Installing the Searchable Plugin	126
Exploring the Searchable Plugin	130
Understanding Services	132
Adding a Search Box	132
Changing the Home Page with UrlMappings	140
Production Deployment Checklist	142
 <b>CONCLUSION</b>	 <b>145</b>
 <b>ABOUT THE AUTHORS</b>	 <b>147</b>

*The proof of the pudding is in the eating. By a small sample we may judge of the whole piece.*

*- Miguel de Cervantes Saavedra*



# 1

## Introduction

---

Grails is an open-source web application framework that's all about getting things done. It uses best of breed technology that most Java developers are already using – most notably Spring and Hibernate – yet somehow Grails is greater than the sum of its parts.

From the moment you type `grails create-app`, you can tell that this isn't your typical Java development project. The fact that everything has its place in Grails – every new component that you add already has a spot waiting for it – gives Grails an oddly familiar feeling, even if you are using it for the first time. It's only after the fact that you realize that you spent your time focusing on *business* solutions instead of *software* solutions.

## Learning by Example

---

This book introduces Grails by example. You'll see how to quickly build a Grails application from scratch and how to customize it to meet various needs.

In order to follow along, you'll need a basic knowledge of object-oriented programming and MVC web application development. You'll certainly benefit from a familiarity with Java, though you can probably get by without it.

You'll also see extensive use of Groovy throughout the examples. While this book doesn't aim to teach Groovy, the examples are such that anyone with some programming background should be able to follow along.

## The RaceTrack Application

---

Over the course of this book, you'll explore the various aspects of Grails development as you build a small web application called *RaceTrack*. There's a regional running club in the southeastern United States that currently uses a paper-based process for tracking the races that the club participates in and the runners that register for each race. They're ready to make the leap into the digital era, and we'll help them do so in short order.

For starters, they want an application that will allow the club's staff members to manage the races and registrations. They assure us they don't need anything too fancy – remember, they're using paper now – so, they'll be happy with an application that provides an internal administrative interface to this data.

Developing the *RaceTrack* application will offer a broad, hands-on exposure to Grails. You'll build a web user interface, manage relationships among database tables, apply validation logic, and develop custom queries. As we continue to expand the application, you'll also explore custom tag libraries, Java integration, security, page layout, and the power of dynamic methods. Before we wrap up, you'll look at unit testing and deployment.

By and large, the information and source code you need to build the application are included within the text of the book. You can download the finished application, but I highly recommend that you build the application yourself as you read along. Trust me when I tell you that, thanks to the power and brevity of Groovy, this is one application that you can easily type in by hand.

The source code for the Racetrack application can be checked out from source control at <http://github.com/scottdavis99/gswg-v2/>. Install Git from <http://git-scm.com/>. Once Git is installed, type `git clone http://github.com/scottdavis99/gswg-v2.git`.

### Free Online Version

Support this work, buy the print copy:

<http://www.infoq.com/minibooks/grails-getting-started>

# 2

## Installing Grails

---

### Installing a JDK

You'll need at least JDK 1.5 in order to work through the examples in this book. Take a moment to download and install it from <http://java.sun.com/javase/downloads>. Then point the `JAVA_HOME` environment variable at your JDK installation.

Type `java -version` to verify that everything is installed correctly.

### Installing Grails

Next, download the latest release of Grails from <http://grails.org/Download>. (This book uses Grails 1.2.)

If you can install the JDK from scratch, then installing Grails is just as simple:

1. Unzip Grails (Note: Be sure that none of the directory names have spaces in them. I use `/opt/grails` on Unix/Linux/Mac OS X machines, or `c:\opt\grails` on Windows machines.)
2. Create a `GRAILS_HOME` environment variable
3. Add `GRAILS_HOME/bin` to the `PATH`

Type `grails` to verify that Grails is installed and ready to use. (For more information on installing Grails, see <http://grails.org/Installation>.)

## Installing a Database

---

Grails ships with an embedded copy of HSQLDB, a pure-Java relational database. HSQLDB is great for quick demos, but at some point you'll most likely want to upgrade to a full-fledged database. Since the Grails Object-Relational Mapping (GORM) API is a thin Groovy façade over Hibernate, any database that has a JDBC driver and a Hibernate dialect is supported.

We'll use MySQL later in this book, but feel free to use another database if it is already installed and configured. You can download a free copy of MySQL Community Edition from <http://dev.mysql.com/downloads/>.



# 3

## Creating a Grails Application

---

In the previous chapter, you installed Grails. In this chapter, you'll create your first Grails application.

You'll see how Grails' scaffolding gets you up and running in a hurry—from laying out a basic directory structure to creating the web pages for basic Create/Read/Update/Delete (CRUD) functionality. Along the way, you'll learn how to change the port that Grails runs on, get introduced to the basic building blocks of a Grails application (domain classes, controllers, and views), specify default values for fields, and much more.

## Creating the RaceTrack Application

---

Now that you have Grails installed, create a directory for your web applications.

```
$ mkdir web
$ cd web
```

All modern Java IDEs offer Groovy and Grails support: IntelliJ, NetBeans, and Eclipse. Text editors like TextMate, vi, and Emacs also have plug-ins for Groovy and Grails. Rather than getting bogged down in the specifics of any particular application, we'll stick to the command line interface.

To create the structure for your first Grails application, type `grails create-app racetrack`.

```
$ grails create-app racetrack
Welcome to Grails 1.2 - http://grails.org/
Licensed under Apache Standard License 2.0
Grails home is set to: /opt/grails
...

Created Grails Application at /web/racetrack
```

You should see a flurry of activity as Grails creates the basic directory structure for your new application.

Even though you haven't created any domain classes or web pages, let's start our application as a quick sanity check. Change to the `racetrack` directory and type `grails run-app`.

```
$ cd racetrack
$ grails run-app
...

Base Directory: /Users/sdavis/web/racetrack
Running script /opt/grails/scripts/RunApp.groovy
Environment set to development
[mkdir] Created dir:
/Users/sdavis/.grails/1.2/projects/racetrack/classes
[groovyc] Compiling 6 source files to
/Users/sdavis/.grails/1.2/projects/racetrack/classes
...

Running Grails application..
Server running. Browse to http://localhost:8080/racetrack
```

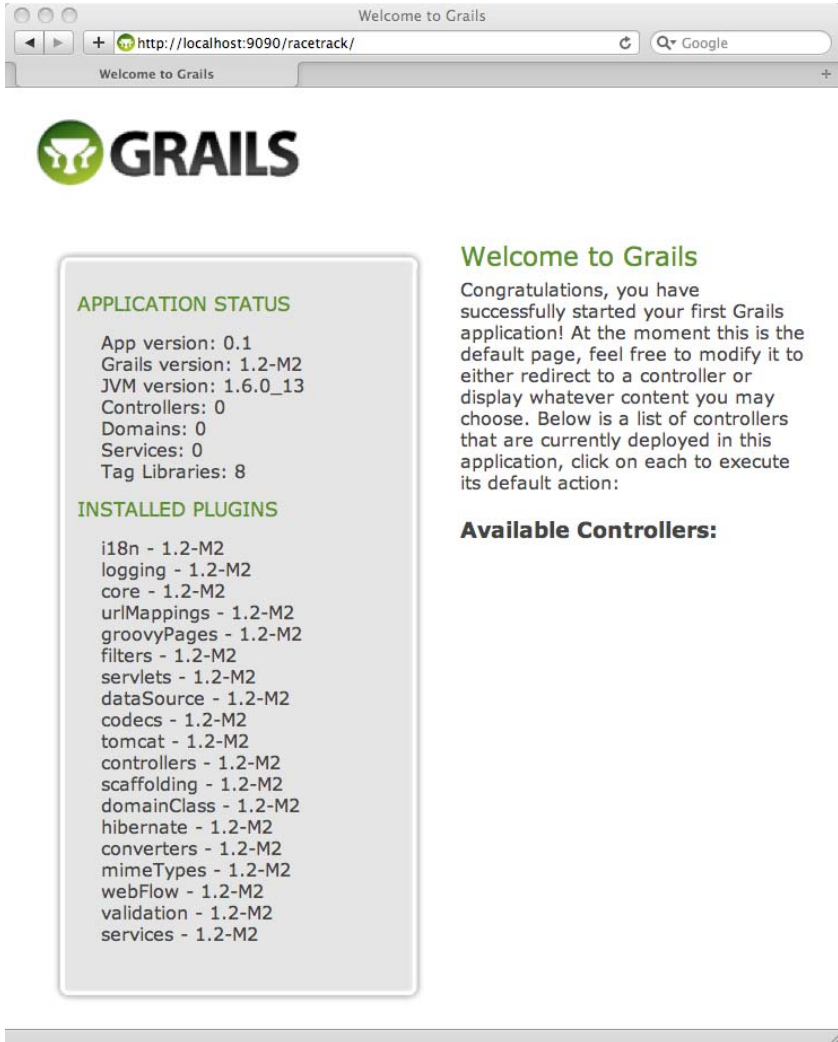
If everything is OK, you should be able to visit <http://localhost:8080/racetrack/> in your web browser and see the welcome screen. If, on the other hand, you already had a service running on port 8080, you were probably greeted with an error message like this:

```
Server failed to start: LifecycleException: Protocol
handler initialization failed:
java.net.BindException: Address already in use<null>:8080
```

Since I already have a Tomcat instance running at port 8080 on my system, I'm going to change the value to 9090 to avoid any future port conflicts. You can change the port permanently in `$GRAILS_HOME/scripts/_GrailsSettings.groovy`. You can also set this value on a case-by-case basis by overriding the value at the command line:

```
$ grails -Dserver.port=9090 run-app
```

Now that I'm running Grails on a clear port, I should be able to see the welcome screen.



Before we move on to actually creating the RaceTrack application, there is one more bit of housekeeping minutiae that you should be aware of. Looking back at the `grails run-app` output, it is rife with references to the `.grails/1.2/projects/racetrack` subdirectory in your home directory. This is where Grails stores all of its temporary files. Notice the leading dot? That keeps it hidden on Unix-like systems. (Those Grails developers are a secretive bunch, aren't they?)

Later in the book, you'll type `grails war` to bundle everything up into a tidy package for deployment into Tomcat, JBoss, or any standard Java Enterprise Edition (JEE) application server. Until then, this is where your

running, generated code can be found. A quick `grails clean` will wipe out this directory, or you can delete it as you would any other directory. You can do this regularly and without fear of repercussions – every time you type `grails run-app`, it will get recreated and repopulated with the latest compiled code.

## The Grails Directory Structure

---

Now that you have a better understanding of the mechanics behind Grails, let's take a closer look at the parts that make up a Grails application.

Grails places a huge emphasis on *convention over configuration*. This means that rather than relying on a *configuration* file to tie the various parts of the application together, Grails relies on *conventions*. All of the domain classes are stored in the `domain` directory. The controllers live in the `controllers` directory, the views in the `views` directory, and so on. Since everything already has a predefined storage location, you are freed from having to write even a single line of configuration.

Figure 3-1 describes how a Grails application is organized.

racetrack	
grails-app	
conf	Configuration files (such as data sources, URL mappings, and legacy Spring and Hibernate configuration files)
controllers	Controllers (The "C" in MVC)
domain	Domain classes (The model or "M" in MVC. Each file in this directory has a corresponding table in the database.)
il8n	Resource bundles (for internationalizing error messages and labels)
services	Service classes (for business logic that potentially spans many domain classes)
taglib	Custom tag libraries (for building reusable elements for use in GSP pages)
utils	Custom scripts (for miscellaneous utilities like codecs)
views	Groovy Server Pages (GSPs) (The "V" in MVC)
lib	JARs (for JDBC drivers and third-party libraries)
scripts	Gant scripts (for project-specific scripts)
src	
groovy	Generic Groovy source files (for files that don't otherwise have a conventional location)
java	Java source files (for pulling in legacy Java code). (Files in this directory will be compiled and included in the WAR file.)
test	
integration	Test scripts that rely on other components (such as databases)
unit	Test scripts that test components in isolation
web-app	
css	Cascading Style Sheets (CSS)
images	Image files (JPGs, GIFs, PNGs, etc.)
js	JavaScript files (including third-party libraries such as Prototype, script.aculo.us, YUI, etc.)
META-INF	The typical JEE directory for manifest files
WEB-INF	The typical JEE directory for web.xml and other configuration files

**Figure 3-1: Application Directory Structure**

## Domain Classes

Domain classes are the lifeblood of a Grails application. In simple terms, they define the “things” that you are trying to keep track of.

Grails takes these simple classes and does quite a bit with them. A corresponding database table is automatically created for each domain class. Controllers and views derive their names from the associated domain class. This is where validation rules are stored, one-to-many relationships are defined, and much much more.

Of course, the typical end user probably describes the application in terms of domain classes as well. “Let’s see, we need to keep track of races and registrations and...”

Take a look at the paper form that the running club uses for each race:

**Southeastern Regional Running Club**

Race Name: TURKEY TROT Distance: 5K

Date: Nov 22, 2007 Time: 9:00 AM

Location: DUCK, NC

Maximum # Runners: 350 Cost: \$20.00

**Registered Runners:**

Date	Name	Address	E-Mail	Gender	DoB
JUNE 1, 2007	JANE DOE	1334 ROCKY ROAD SOMEWHERE, NC 12345	jane@doe.com	F	FEB 1, 1975
JUNE 3, 2007	JOHN DOE	567 SUNDAY DRIVE ELSEWHERE, NC 12345	john@doe.com	M	JAN 1, 1974

**Figure 3-2: Sample Form Used in Current Paper-Based Process**

Can you see the races and registrations? Can you figure out what fields you’ll need to create for each? Experienced software developers probably picked up on the one-to-many relationship between those two entities as well—one race has many registrations.

Let’s create the domain classes. Back at the command prompt, type `grails create-domain-class Race`. (If your application is still running from earlier, press Control-C to stop it.)

```
$ grails create-domain-class Race

. . .

Created Domain Class for Race
Created Tests for Race
```

*NOTE: For simplicity sake, you can ignore the warning about not specifying a package. Or you can type `grails create-domain-class com.acme.Race` to proceed using the package `com.acme`.*

Notice that Grails creates both a domain class and a test class. We'll put the test class to use later. For now, let's focus on the domain class.

Open `racetrack/grails-app/domain/Race.groovy` in your text editor.

```
class Race {
    static constraints = {}
}
```

Hmm... not much to look at so far, eh? Let's add the properties we scavenged from the paper form:

```
class Race {
    String name
    Date startDate
    String city
    String state
    BigDecimal distance
    BigDecimal cost
    Integer maxRunners = 100000

    static constraints = {}
}
```

Each field has a name and a Java datatype. We're setting a default value of 100,000 for the `maxRunners` field. Later, you'll learn how to set up a validation rule that can, among other things, specify a range of legal values for a given field. For right now, though, this is plenty to get us started. (Don't worry about the `static constraints` line for now – you'll see what it's for in the next chapter.)

*Quick Note:* Groovy autoboxes decimal values like 10.5 into a `java.math.BigDecimal` instead of a `java.lang.Float` or a `java.lang.Double` as you might expect. Why is that? For a painful example that not many Java developers are aware of, write a quick Java application that loops 10 times, adding 0.1 to the sum on each iteration. You'll end up with either 0.99999 or 1.000001 depending on whether you stored the sum in a `Double` or a `Float`. Using a `BigDecimal`, you'll get the expected 1.0 each time. This is what the cool kids call "The Principle of Least Surprise."

Let's create the other domain class. At the command prompt, type `grails create-domain-class Registration`. Add the following fields to `racetrack/grails-app/domain/Registration.groovy`:

```
class Registration {
    String name
    Date dateOfBirth
    String gender
    String address
    String city
    String state
    String zipcode
    String email
    Date dateCreated //Note: this is a special name

    static constraints = {}
}
```

There shouldn't be anything too surprising here except for that last field. If you name a `Date` field `dateCreated`, Grails will automatically fill in the value when the instance gets saved to the database for the first time. If you create a second `Date` field named `lastUpdated`, Grails will fill in the date each subsequent time the updated record is saved back to the database.

This convention can be easily disabled via configuration—simply add a `static mapping` block to your class:

```
class Registration {
    // ...

    Date dateCreated
    Date lastUpdated

    static mapping = {
        autoTimestamp false
    }
}
```



On the other hand, if you want Grails to do something more sophisticated than simply fill in a timestamp field or two, you can tap into the lifecycle events of the domain class by creating some intuitively named closures:

```
class Registration {
    // ...

    def beforeInsert = {
        // your code goes here
    }

    def beforeUpdate = {
        // your code goes here
    }

    def beforeDelete = {
        // your code goes here
    }

    def onLoad = {
        // your code goes here
    }
}
```

The static mapping block is technically for mapping class names to alternate table names and field names to alternate column names, but you can do other interesting things with it as well. For example, if you always want lists of Races to be returned in a specific sort order, add this to `Race.groovy`:

```
class Race {

    static mapping = {
        sort "startDate"
    }

    // ...
}
```

For more information on the static mapping block, see <http://grails.org/GORM+-+Mapping+DSL>.

But let's not run before we can walk. (Sorry, no pun intended.) You haven't even seen these simple classes in action. Let's quickly get a Controller and some views in place so that we can take them out for a test drive in the web browser.

## Scaffolding Controllers and Views

---

Controllers and views round out the “Big Three” in Grails applications. The Model/View/Controller (MVC) pattern is undoubtedly familiar to most web developers. By enforcing a clean separation of concerns among these three elements, your efforts are paid back in terms of maximum flexibility and reuse.

The two domain classes we defined earlier – `Race` and `Registration` – are little more than “dumb” placeholders for the data. (This is a bit of an over-simplification, but for right now it is more correct than not.) A domain class isn’t concerned with how it is stored or how it is ultimately displayed to the end user. In the simplest terms, the Controller pulls the data out of the database, creates new Models, and hands them off to the View for display.

We’ll dig deeper into customizing the behavior of Controllers and the look and feel of Views later. For now, let’s let Grails handle the mundane details of scaffolding them both out for us. As you’ll see, getting the default behavior in place is a one-line affair.

Type `grails create-controller Race` at the command prompt.

```
$ grails create-controller Race
. . .
Created Controller for Race
[mkdir] Created dir: /web/racetrack/grails-
app/views/race
Created Tests for Race
```

Notice that in addition to creating `grails-app/controllers/RaceController.groovy`, Grails created a corresponding test. (This is true of most create commands – `create-service`, `create-tag-lib`, and so on.)

Notice, also, that Grails created an empty views directory for the `Race` class. Later on when you type `grails generate-views` so that you can customize the Groovy Server Pages (GSPs), you’ll know exactly where to look for them.

Take a look at your new controller in a text editor:

```
class RaceController {
  def index = { }
}
```

It's just about as empty as your initial domain class, isn't it? There is one crucial difference between domain classes and controllers. While domain classes are responsible for holding the data, Controllers dictate the URLs that the end users ultimately see in their web browser. The empty index closure here is like the `index.jsp` or `index.html` files that you'd set up in other web frameworks – it is the default target for incoming Race requests from the web.

If you wanted to, you could add something like:

```
class RaceController {
  def index = {
    render "Hello World"
  }
}
```

That would greet the end user with a universal message recognized by all software developers. “Hello World” would show up when they visited <http://localhost:9090/racetrack/race> in their browser.

Any closure that you define here is exposed as a URL. For example, you could provide the expected response to <http://localhost:9090/racetrack/race/foo> by rendering “Bar”:

```
class RaceController {
  def index = {
    render "Hello World"
  }

  def foo = {
    render "Bar"
  }
}
```

But we're just getting silly here, aren't we? Let's add some real business value to this exercise. Rather than adding your own custom code, change `RaceController` to look like this:

```
class RaceController {
  def scaffold = Race
}
```

What the heck is that? Actually, it's the code to provide full Create/Read/Update/Delete (CRUD) functionality for the Race class. When Grails sees the `scaffold` property in a controller, it will *dynamically* generate the controller logic and the necessary views for the specified domain class. All from that one line of code!

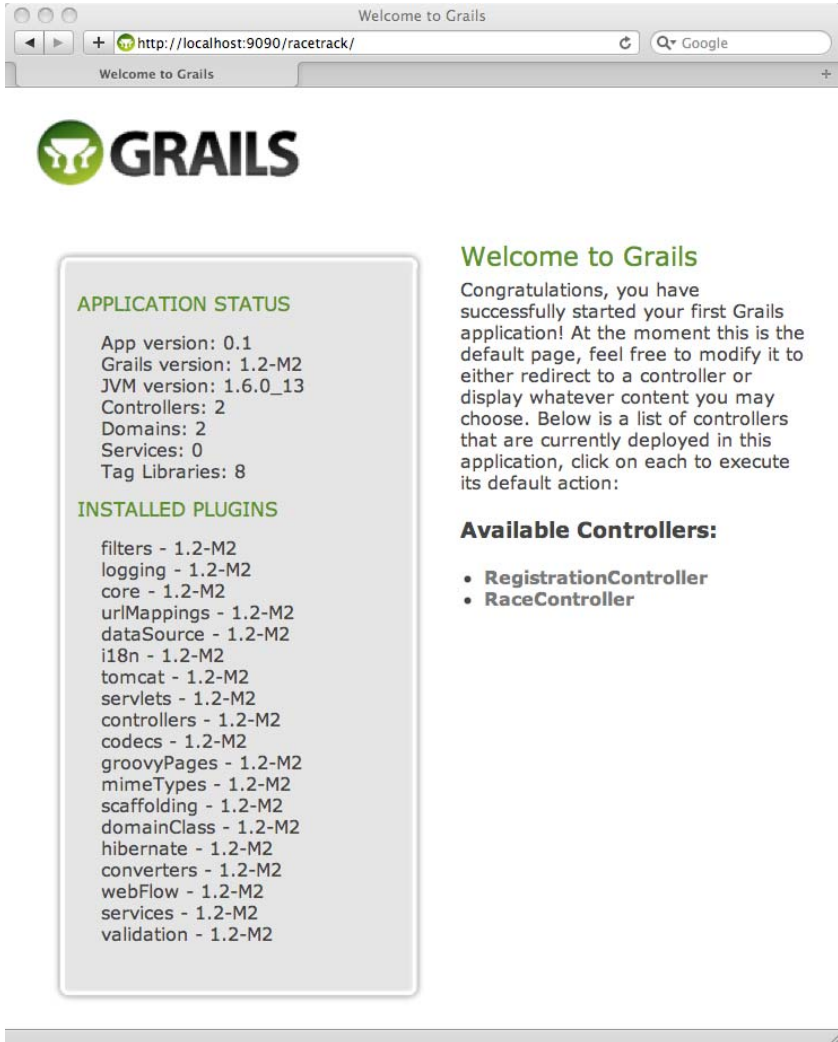
You might also see a more generic form of the scaffold command out in the wild:

```
class RegistrationController {  
    def scaffold = true  
}
```

When you use `true` instead of the domain class name, convention over configuration kicks in once again. Grails looks at the name of the controller class to figure out which domain class to scaffold out.

With both the `RaceController` and `RegistrationController` in place, let's see if all of this scaffolding lives up to our expectations. Can you really have a full CRUD application by just specifying the fields in the domain class and adding one line to the controller? Type `grails run-app` and visit <http://localhost:9090/racetrack> to find out.

The first thing you should see is links for each of the new controllers you created. (Just look for the bulleted list of "Available Controllers.")



Welcome to Grails

http://localhost:9090/racetrack/

Google

**GRAILS**

**APPLICATION STATUS**

App version: 0.1  
 Grails version: 1.2-M2  
 JVM version: 1.6.0\_13  
 Controllers: 2  
 Domains: 2  
 Services: 0  
 Tag Libraries: 8

**INSTALLED PLUGINS**

filters - 1.2-M2  
 logging - 1.2-M2  
 core - 1.2-M2  
 urlMappings - 1.2-M2  
 dataSource - 1.2-M2  
 i18n - 1.2-M2  
 tomcat - 1.2-M2  
 servlets - 1.2-M2  
 controllers - 1.2-M2  
 codecs - 1.2-M2  
 groovyPages - 1.2-M2  
 mimeTypes - 1.2-M2  
 scaffolding - 1.2-M2  
 domainClass - 1.2-M2  
 hibernate - 1.2-M2  
 converters - 1.2-M2  
 webFlow - 1.2-M2  
 services - 1.2-M2  
 validation - 1.2-M2

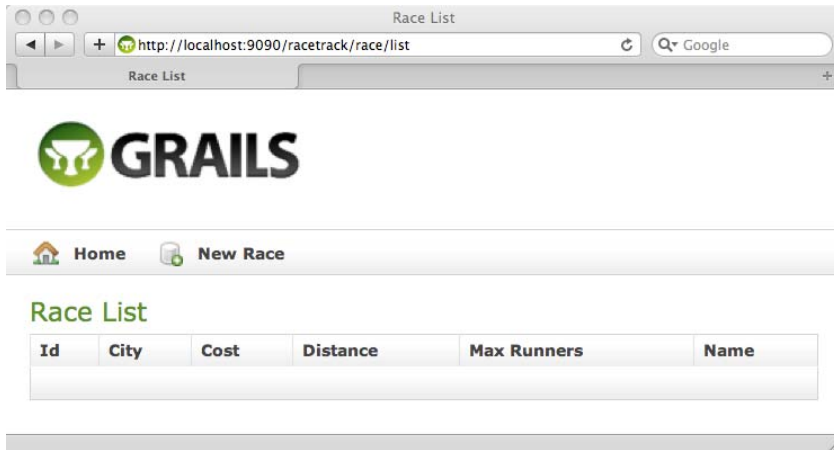
**Welcome to Grails**

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:**

- [RegistrationController](#)
- [RaceController](#)

Click on the `RaceController` link to see a list of existing races.



Grails displays the first six fields of the class, including the `id` field it creates to hold the primary key. Notice how camel-cased field names like `maxRunners` are beautified for the end user?

Now click on `New Race`.

Create Race

http://localhost:9090/racetrack/race/create

Create Race

**GRAILS**

Home Race List

### Create Race

City

Cost

Distance

Max Runners

Name

Start Date     :

State

**Create**

We can see all of the fields of the class in this form. Notice that the default value for `maxRunners` is pre-populated with the value we specified in the domain class.

Go ahead and play around with your shiny new web application—I'll wait right here. Verify that editing and deleting races works as expected. Create a couple of new registrations while you're at it.

Just don't get too attached to any the information you type in. Once you press `Control-C`, all of it will be gone. The next time you type `grails run-app`, you'll have another shiny new (and yes, empty) RaceTrack application waiting for you.

Is this is bug or a feature? Trick question: it's neither; it's convention! You are running Grails in development mode, which means implicitly that you'd like everything to be wiped out between invocations. There are also

test and production modes that each has unique behavior. Type `grails prod run-app`, and your data will have a slightly longer shelf-life. (And don't worry – `grails war` creates a WAR file that runs in production mode.) You'll learn how to change this behavior in the *Databases* chapter.

Oh, and is that field ordering (or lack thereof) bugging you? We will change that in the next chapter.

How do we move from HSQLDB to another database? How do we set up custom validation? How do we define one-to-many relationships between our domain classes?

These are all good questions. I'm sure that you have many more. But once again, let's not get ahead of ourselves. You have a full race to run, and you're just barely out of the starting blocks. There is certainly plenty more to do, but don't lose sight of what you've already accomplished.

Type `grails stats` at the command line:

```
$ grails stats
Welcome to Grails 1.2 - http://grails.org/
Licensed under Apache Standard License 2.0
Grails home is set to: /opt/grails

Base Directory: /web/racetrack
Running script /opt/grails/scripts/Stats.groovy
Environment set to development
```

Name	Files	LOC
Controllers	2	6
Domain Classes	2	22
Unit Tests	4	44
Totals	8	72

Ignoring the unit tests (which we won't do for very much longer, by the way) you have written 28 lines of code spread between two controllers and two domain classes. In return, you have a full CRUD application up and running. I'd say that you've gotten a pretty decent return on your investment. Wouldn't you agree?



### Free Online Version

Support this work, buy the print copy:

<http://www.infoq.com/minibooks/grails-getting-started>

# 4

## Validation

In the previous chapter, you were introduced to the basic building blocks of a Grails application: domain classes, controllers, and views. In this chapter, we'll dig deeper into domain classes.

You'll learn how to specify the order of fields. You'll explore the validation options that Grails offers out-of-the-box, and create your own custom validation. Of course, once you create a custom validation, you'll want to know how to test it as well. And finally, you'll see how to change the error messages your app provides for validation errors.

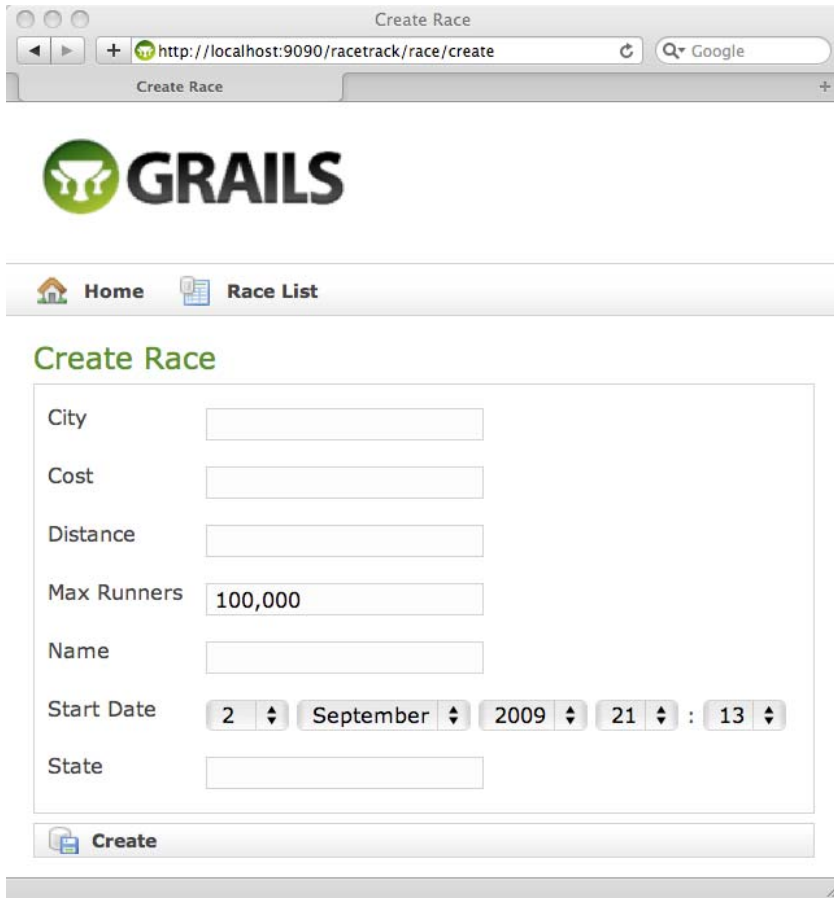
## Customizing the Field Order

Let's revisit the Race class. Pull it up in a text editor.

```
class Race {
    String name
    Date startDate
    String city
    String state
    BigDecimal distance
    BigDecimal cost
    Integer maxRunners = 100000

    static constraints = {}
}
```


You added the fields in a specific order – name, startDate, city... Were you surprised when Grails didn't display the fields in the same order?




Create Race

http://localhost:9090/racetrack/race/create

Create Race

 **GRAILS**

 **Home**  **Race List**

### Create Race

City

Cost


Distance

Max Runners

Name

Start Date     :

State

 **Create**

This is a bit of Grails' Java underpinnings showing through – there is no command in Java that says, “Show me the fields of that class in order.” The getter/setter paradigm makes Java classes conceptually more like HashMaps than LinkedLists.

Thankfully, Grails makes it easy to say, “Hey, when you are scaffolding out the web pages for me, please make sure that the fields show up in this particular order.” You give this hint to Grails in the form of a `static constraints` block.

```
class Race {
    static constraints = {
        name()
        startDate()
        city()
        state()
        distance()
        cost()
        maxRunners()
    }

    String name
    Date startDate
    String city
    String state
    BigDecimal distance
    BigDecimal cost
    Integer maxRunners = 100000
}
```

Type `grails run-app` and visit <http://localhost:9090/race-track/race/create> to verify that the fields now appear in the correct order.

Create Race

http://localhost:9090/racetrack/race/create

Create Race

**GRAILS**

**Home** **Race List**

### Create Race

Name

Start Date

City

State

Distance

Cost

Max Runners

**Create**

Now I know what you are thinking: “Why should I have to type everything in twice? Doesn’t that violate the Don’t Repeat Yourself (DRY) principle?” I’d agree with you if that was all that the `static constraints` block did. But in fact, field ordering is more of a happy side effect than anything else. The real reason you add it is for validating (or constraining) the fields.

## Adding Validation

Talking to the race organizers, many constraints come up in casual conversation:

- “The name of a race can’t be more than 50 letters, or else it won’t fit on the t-shirts and banners.”
- “We’re a regional club. We only run races in Georgia, Virginia, and North and South Carolina.”
- “We’ve never charged more than \$100 for a race, but we’ve done plenty for free as well.”

And as an experienced programmer, you can probably come up with some absurd but necessary constraints that the end user might never have considered:

- “Every race must have a name.”
- “There’s no such thing as a negative distance for a race.”

With this information in hand, let’s add some reasonable constraints to the Race class:

```
class Race {
    static constraints = {
        name(blank:false, maxSize:50)
        startDate()
        city()
        state(inList:["GA", "NC", "SC", "VA"])
        distance(min:0.0)
        cost(min:0.0, max:100.0)
        maxRunners(min:0, max:100000)
    }

    String name
    Date startDate
    String city
    String state
    BigDecimal distance
    BigDecimal cost
    Integer maxRunners
}
```

And now for the fun part. You’re not being ornery if you try to break your own validation rules – you’re being a conscientious, thorough software engineer. Type in some values (like `-1`) that you know will trigger some validation errors.

The screenshot shows a web browser window titled 'Create Race' with the URL `http://localhost:9090/racetrack/race/save`. The Grails logo is at the top. Below it is a navigation bar with 'Home' and 'Race List' links. The main heading is 'Create Race'. A red-bordered box contains four error messages:

- Property [cost] of class [class Race] with value [-1] is less than minimum value [0]
- Property [distance] of class [class Race] with value [-1] is less than minimum value [0]
- Property [maxRunners] of class [class Race] with value [-1] is less than minimum value [0]
- Property [name] of class [class Race] cannot be blank

Below the errors is the form with the following fields:

- Name:
- Start Date: 2 September 2009 21 : 21
- City:
- State: GA
- Distance: -1
- Cost: -1
- Max Runners: -1

At the bottom is a 'Create' button.

Ah, look at that sea of red! Only a developer is happy when they see an error message – an *expected* error message, that is.

Validation seems to be working. Some of the constraints we set up – `maxSize`, `inList` – make it impossible for the user to provide bogus values (assuming he stays within the confines of the browser and doesn't resort to any kind of hackery). At this point, you can't type more than 50 characters into the name field or choose anything but one of the four valid states. But you can leave fields blank, and you can type incorrect values into text fields. Our validation rules make sure that those mistakes – innocent or intentional – don't pollute the database.

See Figure 4-1 for a full list of available Grails validation options.

Constraint	Usage	Description
blank, nullable	blank:true nullable:true	blank traps for blanks at the web tier; nullable traps for blanks at the database tier
creditCard	creditCard:true	Based on the Apache Commons CreditCardValidator, this guarantees that credit card numbers are internally consistent
display	display:false	Hides the field in create.gsp and edit.gsp. Note: be sure to couple this with blank:true and nullable:true or populate these values in the controller; otherwise the default validation will fail
email	email:true	Ensures that values match the basic pattern "user@somewhere.com"
password	password:true	Changes the view from <input type="text"> to <input type="password">
inList	inList:["A", "B", "C"]	Creates a combo box of possible values
matches	matches:"[a-zA-Z]+"	Allows you to provide your own regular expression
min, max	min:0, max:100 min:0.0, max:100.0	Used for numbers and classes that implement java.lang.Comparable
minSize, maxSize, size	minSize:0, maxSize:100, size:0..100	Used to limit the length of Strings and arrays
notEqual	notEqual:"Foo"	As the name suggests, the value cannot equal the constraint
range	range:0..10	Creates a combo box with an element for each value in the range
scale	scale:2	Sets the number of decimal points
unique	unique:true	Ensures that the value doesn't already exist in the database table. (This is perfect for creating new logins.)
url	url:true	Ensures that the value matches the basic pattern "http://www.somewhere.com"
validator	validator: {return(it%2)==0}	Allows you to create your own custom validation closure

Figure 4-1: Grails Validation Options

## Changing Error Messages

---

The race organizers are pleased that we are scrubbing their data input for them, but one of them said, “It sounds like I’m being scolded by a robot.” Thankfully, changing the default error message is just as easy as setting up the validation in the first place.

Grails stores all error messages in the `messages.properties` file in the `grails-app/i18n` directory. Yes, this is the same Resource Bundle that Java developers are used to using for internationalizing applications. As you look in the `i18n` directory, notice that the standard Grails error messages have already been translated to German, Spanish, French, Italian, Japanese, and many more.

As you browse the file, can you find the template for the blank error message?

```
default.blank.message=Property [{0}] of class [{1}]
cannot be blank
```

Looking back at the error message in the web browser, it looks like `{0}` is a placeholder for the name of the field. The name of the class is stored in `{1}`.

There are a couple of other placeholders that get used in more involved messages. For example, check out the error message for a range violation:

```
default.invalid.range.message=Property [{0}] of class
[{1}] with value [{2}] does not fall within the valid
range from [{3}] to [{4}]
```

The value that the user typed in is stored in `{2}`. The start and end values for the range are stored in `{3}` and `{4}`.

Rather than changing the default error messages, why don’t we set up some custom error messages on a per-class, per-field basis? Add the following to `messages.properties`. (Note that each message should be on a single line.)



```

race.name.blank=Please provide a Name for this Race
race.name.maxSize.exceeded=Sorry, but a Race Name can't
be more than {3} letters
race.distance.min.notmet=A Distance of {2}? What are you
trying to do, run backwards?
race.maxRunners.min.notmet=Max Runners must be more than
{3}
race.maxRunners.max.exceeded= Max Runners must be less
than {3}
race.cost.min.notmet=Cost must be more than {3}
race.cost.max.exceeded=Cost must be less than {3}

```

Try to save your bogus values in the web browser once again. This time, there should be no more “robot scolding” going on:

Create Race

Home Race List

### Create Race

❗ Cost must be more than 0

❗ A Distance of -1? What are you trying to do, run backwards?

❗ Max Runners must be more than 0

❗ Please provide a Name for this Race

Name

Start Date     :

City

State

Distance

Cost

Max Runners

Create

See Figure 4-2 for a full list of available validation error messages.

Constraint	Per-class, Per-field Message
blank,	ClassName.propertyName.blank
nullable	ClassName.propertyName.nullable
creditCard	ClassName.propertyName.creditCard.invalid
display	N/A
email	ClassName.propertyName.email.invalid
password	ClassName.propertyName.password.invalid
inList	ClassName.propertyName.not.inList
matches	ClassName.propertyName.matches.invalid
min, max	ClassName.propertyName.min.notmet ClassName.propertyName.max.exceeded
minSize,	ClassName.propertyName.minSize.notmet
maxSize,	ClassName.propertyName.maxSize.exceeded
size	ClassName.propertyName.size.toosmall ClassName.propertyName.size.toobig
notEqual	ClassName.propertyName.notEqual
range	ClassName.propertyName.range.toosmall ClassName.propertyName.toobig
scale	N/A
unique	ClassName.propertyName.unique
url	ClassName.propertyName.url.invalid
validator	ClassName.propertyName.validator.invalid

**Figure 4-2: Grails Validation Error Messages**

## Creating Custom Validations

Here's an interesting challenge. The race organizers get burned every January by people who inadvertently write in *last year* instead of *this year*. "Is there a way that you can make sure that we don't try to schedule a race in the past? It messes up our books and then our accountants yell at us."

Knowing what you already know about Grails validation, you might be tempted to use a min constraint on the `startDate` field:

```
class Race {
    static constraints = {
        // ...

        // NOTE: This doesn't do
        // what you think it does
        startDate(min: new Date())

        // ...
    }

    Date startDate
    // ...
}
```

While this is a good idea in theory, it doesn't do what you think it does. In this case, the value for `new Date()` gets set each time the server gets restarted, not each time the validation gets evaluated. Conceptually, this is not much different than hard coding a fixed value like we did for `max` and `min` on `distance`, `cost`, and `maxRunners`.

Think of it this way: you want the validation to be the result of an expression, not a fixed value. To accomplish this, you need to create a custom validation. The following code will prevent the race organizers from entering races in the past:

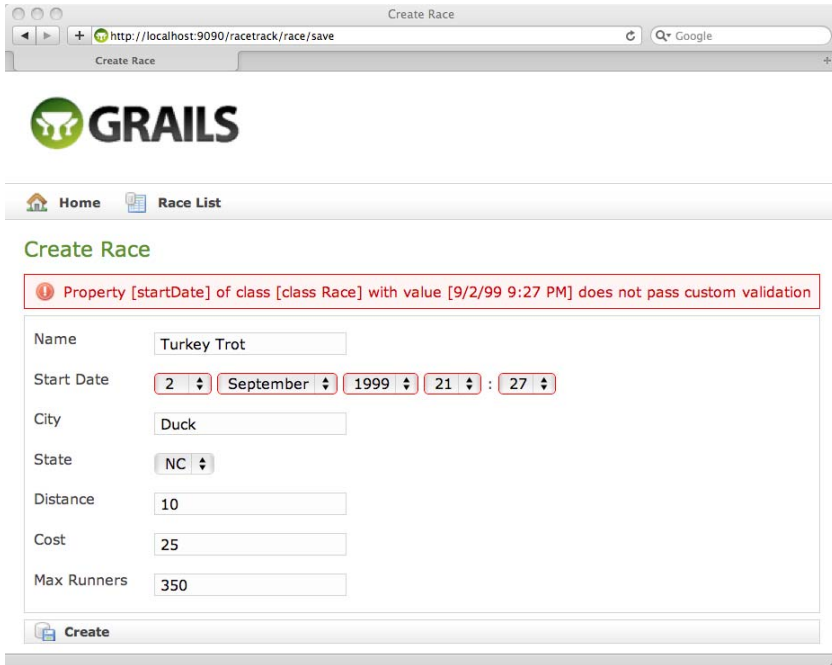
```
class Race {
  static constraints = {
    // ...
    startDate(validator: {return (it > new Date())})
    // ...
  }

  Date startDate
  // ...
}
```

All a custom validator has to do is return a Boolean value. The closure can use as many lines as it needs to return `true` or `false`, as long as it answers the question, “Is this valid?”

The validator closure is evaluated every time it is invoked. This means that `new Date()` is refreshed every time. And in case it isn't obvious, it represents the date that the user is trying to submit.

Try entering a start date for a race in the past:



The screenshot shows a web browser window titled "Create Race" with the URL `http://localhost:9090/racetrack/race/save`. The Grails logo is at the top, followed by navigation links for "Home" and "Race List". The main heading is "Create Race". A red error message box at the top of the form states: "Property [startDate] of class [class Race] with value [9/2/99 9:27 PM] does not pass custom validation". The form fields are as follows:

Name	Turkey Trot
Start Date	2 September 1999 21 : 27
City	Duck
State	NC
Distance	10
Cost	25
Max Runners	350

At the bottom of the form is a "Create" button.

Once you are convinced that the validation is working, make a quick change to `messages.properties` as well: (Remember, this needs to be all on one line.)


```
race.startDate.validator.invalid=Sorry, but the past is  
the past. You should be scheduling races in the future.
```

One more bad date, and one more friendly error message:

Create Race


[http://localhost:9090/racetrack/race/save](#)

Create Race



[Home](#)
[Race List](#)

### Create Race

 **Sorry, but the past is the past. You should be scheduling races in the future.**

Name

Start Date

:


City

State

Distance

Cost

Max Runners

 **Create**

## Testing Validations

---

I've been threatening to write some tests since we started the RaceTrack application. The time has finally come.

Why now? Because we've finally added some custom functionality worth testing. Everything before the "races can't happen in the past" story simply leveraged existing Grails functionality. There's not much value in testing out-of-the-box behavior – let's leave testing core Grails to the Grails development team. We'll focus our efforts on testing our own custom code.

The "monkey-see, monkey-do" testing we've done up to this point – typing in bad values and looking at the results on screen – is OK, but it's hardly what you could call *rigorous*. Of course you should sanity check your work by actually clicking around in the running web application, but your job as a programmer doesn't end there. Seasoned developers know better than to fall into the "but it works on my box" trap.

Writing a quick test *in code* brings two immediate benefits to the table: you've automated the clicks so that you don't have to do them manually every single time you change the code, and you've written a bit of ***executable documentation***. Future developers (or even *you* in a couple of months) will be able to quickly glance at the test and figure out what your intent was.

Recall that Grails created a corresponding unit test for you when you typed `grails create-domain-class Race`. Look in `test/unit` for `RaceTests.groovy`:

```
import grails.test.*

class RaceTests extends GrailsUnitTestCase {
    protected void setUp() {
        super.setUp()
    }

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

    void testSomething() {

    }
}
```

That empty stub of a test is just waiting for you to do something clever with it. For example, add a convenience method to the `Race` class that converts the distance from kilometers to miles.

```
class Race {
    // ...
    BigDecimal distance

    BigDecimal inMiles() {
        return distance * 0.6214
    }
}
```

Now change the generic `testSomething()` method in `RaceTests.groovy` to something a little more useful:

```
import grails.test.*

class RaceTests extends GrailsUnitTestCase {
    // ...
    void testInMiles() {
        def race = new Race(distance:5.0)
        assertEquals 3.107, race.inMiles()
    }
}
```

The first line of `testInMiles()` instantiates a new `Race` and initializes the distance to `5.0`. The next line asserts that 5 km equals 3.107 miles. If the assertion fails, the test will fail as well.

If you are familiar with the Java testing framework JUnit, you'll be pleased to know that a `GrailsUnitTestCase` extends a `GroovyTestCase`, which in turn extends a `JUnit TestCase`. This means that all of the assertions you have come to know and love – `assertTrue`, `assertFalse`, `assertNull`, `assertNotNull`, and so on – are all available to you here as well. (We'll talk more about the various types of `TestCases` in the *Security* chapter.)

To run your tests, type `grails test-app` at the command line.

```
$ grails test-app
. . .

Environment set to test

-----
Running 4 Unit Tests...
Running test RaceControllerTests...PASSED
Running test RaceTests...PASSED
Running test RegistrationControllerTests...PASSED
Running test RegistrationTests...PASSED
Unit Tests Completed in 463ms ...
-----

No tests found in test/integration to execute ...

Tests PASSED - view reports in target/test-reports
```

There's a lot of information in the command output. Let's go through it step-by-step.

1. The first thing you should notice is that the environment is set to test instead of development. (You'll learn much more about environments when we talk about `DataSources`.)
2. Next, you can see that each of the four tests ran and passed, including your new `testInMiles()` test.
3. There were no integration tests found, but that is to be expected – we haven't written any yet. In just a moment, we'll write an integration test for the custom validation.
4. And finally, there is a newly created JUnit report waiting for us in the `target/test-reports/html` directory. Open it in a web browser:



Unit Test Results.

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

### Summary

Tests	Failures	Errors	Success rate	Time
4	0	0	100.00%	0.675

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

### Packages

Name	Tests	Errors	Failures	Time(s)	Time Stamp	Host
<none>	4	0	0	0.675	2009-09-03T03:32:44	scott-davis2s-macbook-pro.local

If any of your tests fail, you can drill down to the class and individual test to see the details.

If you want to focus on a specific class, it's nice to know that you can type `grails test-app Race` to run just that test script alone. (Just leave the “Tests” suffix off of the end of any test script to run it with the `test-app` command.)

Congratulations! You just wrote your first Grails unit test. But didn't we start out looking to test our custom validation? We're going to do that right now, but there is one more important concept you need to learn – the difference between a unit and integration test.

Grails creates unit tests for you because they are the simplest possible tests to run. A unit test doesn't require the web server to be up and running, the database to be initialized, or anything else. A unit test is meant to test the individual *unit* of code in isolation.

An integration test, on the other hand, is a test that needs everything to be up and running. Since our custom validation could rely on a database lookup, a web services call, or anything else, Grails treats it as a whole different category of test.

Create a file named `RaceIntegrationTests.groovy` in `test/integration`.

```
class RaceIntegrationTests extends GroovyTestCase {

    void testRaceDatesBeforeToday() {
        def lastWeek = new Date() - 7
        def race = new Race(startDate:lastWeek)

        assertFalse "Validation should not succeed",
            race.validate()
        assertTrue "There should be errors",
            race.hasErrors()
    }
}
```

Notice that you can call `race.validate()` to test your validations without actually trying to save the class to the database. If everything checks out, it will return `true`. Another way to see if there are problems is to call `race.hasErrors()` after you call `race.validate()`.

You might be tempted to leave the test as it is, but notice that you aren't looking for a specific error. You might get lulled into a false sense of security if you have multiple validations in place as we do. Here is a more detailed test that drills down and asserts that the exact validation error we were expecting is actually getting triggered.

```
class RaceIntegrationTests extends GroovyTestCase {
    void testRaceDatesBeforeToday() {
        def lastWeek = new Date() - 7
        def race = new Race(startDate:lastWeek)
        assertFalse "Validation should not succeed",
            race.validate()
        // It should have errors after validation fails
        assertTrue "There should be errors",
            race.hasErrors()

        println "\nErrors:"
        println race.errors ?: "no errors found"

        def badField = race.errors.getFieldError('startDate')
        println "\nBadField:"
        println badField ?: "startDate wasn't a bad field"
        assertNotNull
            "Expecting to find an error on the startDate field",
            badField

        def code = badField?.codes.find {
            it == 'race.startDate.validator.invalid'
        }
        println "\nCode:"
        println code ?:
            "the custom validator for startDate wasn't found"
        assertNotNull "startDate field should be the culprit",
            code
    }
}
```



Wow, 7 validation errors? Yup – the name was blank, the cost was less than zero, and so on. Even though the second version of our test was more verbose, it was also more accurate.

Well, let's type `grails stats` once again and see where we stand:

```
$ grails stats
```

Name	Files	LOC
Controllers	2	6
Domain Classes	2	33
Unit Tests	4	46
Integration Tests	1	19
Totals	9	104

We're just over 100 lines of code, but we've accomplished quite a bit. We now have a set of web pages that display the fields in the correct order. We have validation in place that scrubs the user input so that no rogue data makes its way into the database. And we've even got a few interesting tests in place, both unit and integration.

## Free Online Version

Support this work, buy the print copy:

<http://www.infoq.com/minibooks/grails-getting-started>

# 5

## Relationships

In the last chapter, you explored validation. In this chapter, we'll shift our focus to the relationships between domain classes.

We'll establish a one-to-many (1:M) relationship between the `Race` and `Registration` classes. We'll then create a many-to-many (M:M) relationship. And finally, we'll bootstrap some data so that we don't have to keep retyping it over and over again.

### Creating a One-to-Many Relationship

When writing the user stories for the RaceTrack application, one of the first things we recognized was the relationship between races and registrations. It wouldn't be much of a race if it didn't have more than one registration, don't you agree?

Take a look at the paper form once again:

**Southeastern Regional Running Club**

Race Name: TURKEY TROT Distance: 5K  
Date: Nov 22, 2007 Time: 9:00 AM  
Location: Duck, NC  
Maximum # Runners: 350 Cost: \$20.00

**Registered Runners:**

Date	Name	Address	E-Mail	Gender	DoB
JUNE 1, 2007	JANE DOE	134 ROCKY ROAD SOMEWHERE, NC 12345	jane@doe.com	F	FEB 1, 1975
JUNE 3, 2007	JOHN DOE	567 SUNDAY DRIVE ELSEWHERE, NC 12345	john@doe.com	M	JAN 1, 1974

Figure 5-1: Sample Form Used in Current Paper-Based Process

Modeling this one-to-many relationship with Grails couldn't be easier – it's almost plain English.

Open `grails-app/domain/Race.groovy` in a text editor. Add a single line to it:

```
class Race {
    // ...
    static hasMany = [registrations:Registration]
}
```

That one line of code creates a new `java.util.Set` field named `registrations`. If `Race` had a bunch of relationships, you could add them one after the other, separated by commas:

```
class Race {
    // ...
    static hasMany = [registrations:Registration,
                     locations:Location, sponsors:Company]
}
```

Are you beginning to see a pattern emerge? That same `name:value` list is what you use to define constraints, initialize values in the constructor, and so on. Here, the name is the name of the field and the value is the datatype. In the case of `sponsors` and `Company`, do you see how you can easily make the name descriptive and independent of the class name?

If you stopped here, you'd have a uni-directional one-to-many relationship. In other words, a `Race` would know about its registrations, but a `Registration` wouldn't know which `Race` it belonged to.

If you want to make the relationship bi-directional, simply add a single line of code to the “many” end of this one-to-many relationship.

```
class Registration {
    // ...
    static belongsTo = [race:Race]
}
```

This not only closes the loop; it also enforces cascading updates and deletes.

With our whopping two lines of code in place, let's see it in action. Create a new race:

Browser: Show Race  
 Address: http://localhost:9090/racetrack/race/show/1  
 Search: Google

**GRAILS**

Home Race List New Race

## Show Race

*i* Race 1 created

<b>Id:</b>	1
<b>Name:</b>	Turkey Trot
<b>Start Date:</b>	2009-11-02 21:47:00.0
<b>City:</b>	Duck
<b>State:</b>	NC
<b>Distance:</b>	10
<b>Cost:</b>	25
<b>Max Runners:</b>	350
<b>Registrations:</b>	

Edit
 Delete

Notice the new *Registrations* field? It doesn't appear when you are first creating the race since the "one" side of the relationship quite literally doesn't exist yet. But once you save the race, you can begin adding registrations to the "many" side of the relationship.

Create Registration

http://localhost:9090/racetrack/registration/create

Create Registration

**GRAILS**

Home Registration List

### Create Registration

Address

City

Date Created     :

Date Of Birth     :

Email

Gender

Name

Race

State

Zipcode

Create

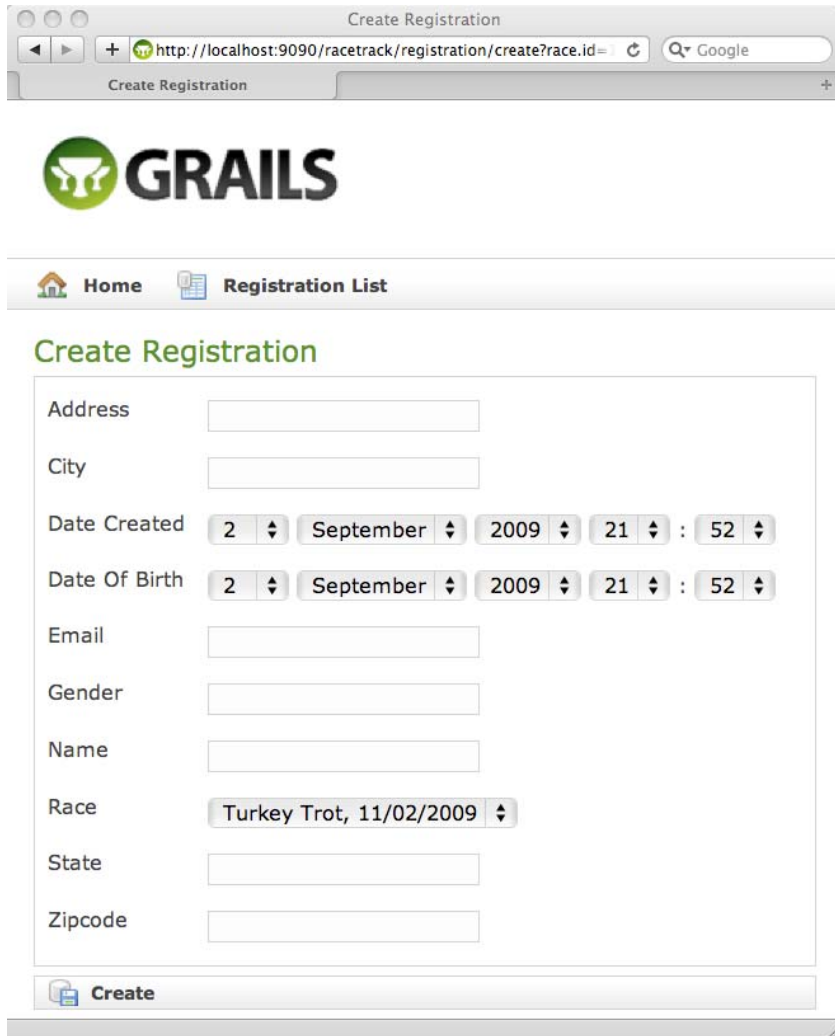
Notice the combo box for specifying the race associated with this registration? In the chapter on validation, we used `inList` to create an arbitrary list of values. This time, thanks to `hasMany` and `belongsTo`, the list is populated from a database table. And all the end user has to do is memorize the primary keys from the other table. That's not too much to ask in exchange for an automatic solution, is it?



Wait, it is? OK, then you'll need to make one more tiny tweak to `Race.groovy`. Grails uses the name of the class and the primary key for the combo box unless there is a `toString()` method. (Yes, Java developers, the same `toString()` method that you expect to see as well.)

```
class Race {  
    // ...  
    static hasMany = [registrations:Registration]  
  
    String toString(){  
        return "${name}, ${startDate.format('MM/dd/yyyy')}"  
    }  
}
```

The `toString()` method doesn't have to return a unique value – that's the job of the primary key – but in our case, the name of the race and the `startDate` do a great job of making the items in the combo box both unique and human-readable.



Create Registration

Home Registration List

### Create Registration

Address

City

Date Created 2 September 2009 21 : 52

Date Of Birth 2 September 2009 21 : 52

Email

Gender

Name

Race Turkey Trot, 11/02/2009

State

Zipcode

Create

So, with a `hasMany` declaration, a `belongsTo` declaration, and a `toString()` method, we've managed to create a one-to-many relationship. But do you know what? I think that we missed a class and a relationship the first time around. I think that we actually have a many-to-many relationship right under our nose.

## Creating a Many-to-Many Relationship

The race organizers are understandably focused on races and registrations – that’s what they do. But what about the runners? More specifically, what about the runners who run more than one race with us? One race can have many runners, but one runner can have many races as well. It looks like we have a classic many-to-many relationship on our hands. (This is what the cool kids call *emergent design*.)

If, as a runner, you sign up for a dozen races a year, you are going to get pretty tired of typing in your address, gender, and email address over and over again for each race. Why don’t we capture that information once and reuse it each time you register for a new race?

You already know how to create a new domain class. Type `grails create-domain-class Runner` at the command prompt. Move the appropriate fields from `Registration` over to `Runner`. Here’s what the two classes should look like after you’re done:

```
class Runner {
    static constraints = {
        firstName(blank:false)
        lastName(blank:false)
        dateOfBirth()
        gender(inList:["M", "F"])
        address()
        city()
        state()
        zipcode()
        email(email:true)
    }

    static hasMany = [registrations:Registration]

    String firstName
    String lastName
    Date dateOfBirth
    String gender
    String address
    String city
    String state
    String zipcode
    String email

    String toString(){
        "${lastName}, ${firstName} (${email})"
    }
}
```

```

class Registration {
    static constraints = {
        race()
        runner()
        paid()
        dateCreated()
    }

    static belongsTo = [race:Race, runner:Runner]

    Boolean paid
    Date dateCreated
}

```

Notice that once we extracted all of the stuff that didn't really pertain to a `Registration`, we discovered another emerging requirement? We forgot to check whether the runner has paid for the race or not. (The paper form doesn't have a checkbox for this since you can't sign up if you don't have cash in hand.)

Many-to-many relationships are one of the most frequently misunderstood relationships in software development. If we had started our analysis with `Runners` and `Races`, we probably would have figured out that they share a M:M relationship early on. But experienced software developers don't stop there—the good ones keep digging around like a paleontologist until they've unearthed the hidden third class (`Registration`, in our case). In other words, a M:M relationship is really just two 1:M relationships with a third class that you haven't discovered yet.

Let's see what our new class looks like in a web browser. But before you type `grails run-app` again, don't forget to create a new controller for managing runners—`grails create-controller Runner`.

```

class RunnerController {
    def scaffold = true
}

```

Change the new controller so that it scaffolds everything out. You should now be able to create a new race ...

Browser address bar: `http://localhost:9090/racetrack/race/show/1`

**GRAILS**

Home Race List New Race

## Show Race

**Race 1 created**

Id:	1
Name:	Turkey Trot
Start Date:	2009-11-02 21:47:00.0
City:	Duck
State:	NC
Distance:	10
Cost:	25
Max Runners:	350
Registrations:	

Edit Delete

... create a new runner ...




Show Runner

http://localhost:9090/racetrack/runner/show/1

Google

Show Runner



# GRAILS

 **Home**  **Runner List**  **New Runner**

## Show Runner

 **Runner 1 created**

<b>Id:</b>	1
<b>First Name:</b>	Jane
<b>Last Name:</b>	Doe
<b>Date Of Birth:</b>	1980-01-01 21:58:00.0
<b>Gender:</b>	F
<b>Address:</b>	123 Main St
<b>City:</b>	Goose
<b>State:</b>	NC
<b>Zipcode:</b>	12345
<b>Email:</b>	jane@whereever.com
<b>Registrations:</b>	

 **Edit**  **Delete**

... and then register the runner for a race:

Create Registration

Home Registration List

### Create Registration

Race: Turkey Trot, 11/02/2009

Runner: Doe, Jane (jane@whereever.com)

Paid: ☐

Date Created: 2 September 2009 22:00

Create

## Bootstrapping Data

Are you getting tired of retyping your data each time you restart Grails? Remember this is a feature, not a bug. We're in development mode, and that means that the tables are created every time you start Grails and dropped every time you shut down. You'll learn how to adjust that feature in the *Databases* chapter, but until then, here's something that you'll find helpful regardless of what mode you're running in.

Look in the `grails-app/conf` directory for a file named `BootStrap.groovy`.

```
class BootStrap {
    def init = { servletContext ->
    }

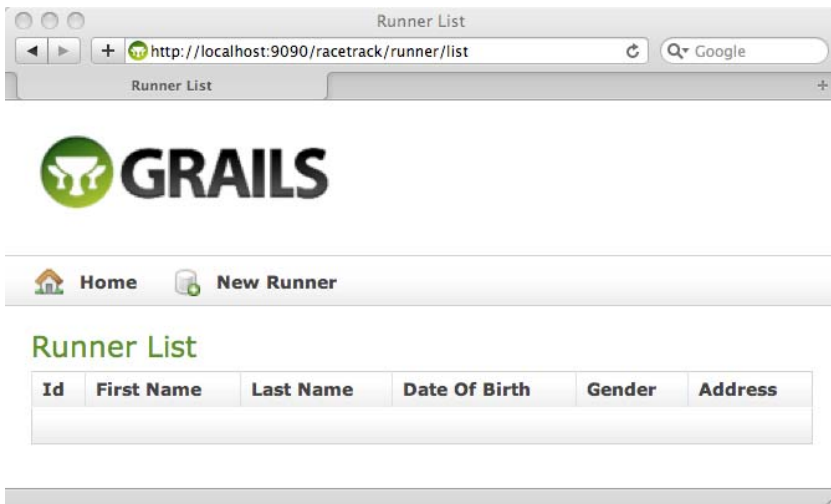
    def destroy = {}
}
```

As you might have already guessed, the `init` closure is called every time Grails starts. The `destroy` closure is called when Grails shuts down.

Do you remember how we instantiate a class in a unit test? We'll use the same trick here, only this time we'll save the class to the database as well.

```
class BootStrap {
  def init = { servletContext ->
    def jane = new Runner(firstName:"Jane",
                          lastName:"Doe")
    jane.save()
  }
  def destroy = {}
}
```

Fire up Grails and go to <http://localhost:9090/racetrack/runner/>:



Hmm, Jane didn't make it into the database. Any thoughts on how to see what the problem is?



```

class BootStrap {
  def init = { servletContext ->
    def jane = new Runner(firstName:"Jane",
                          lastName:"Doe")

    jane.save()
    if(jane.hasErrors()){
      println jane.errors
    }
  }

  def destroy = {}
}

```

Of course! If it worked in the test, it'll work here as well, right? Fire up Grails once again.

```

$ grails run-app
. . .

Environment set to development
Running Grails application..

org.springframework.validation.BeanPropertyBindingResult:
7 errors
Field error in object 'Runner' on field 'address':
rejected value [null];

. . .

```

Wow, those constraints are really picky, aren't they? Let's construct a well-formed runner this time:

```

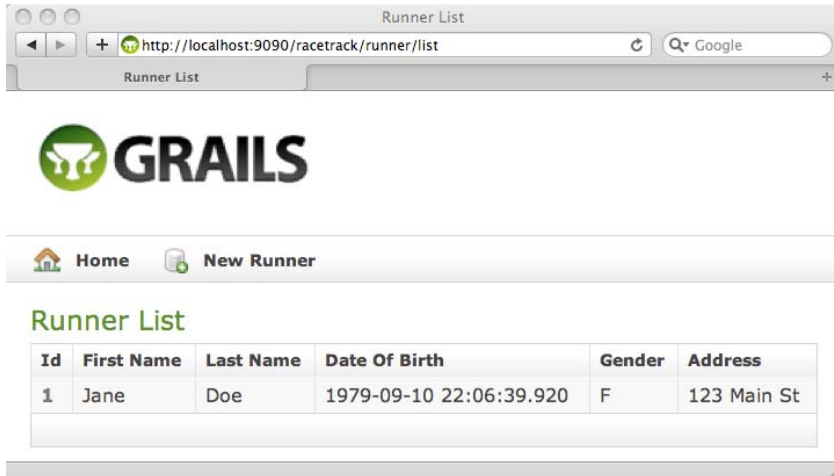
class BootStrap {
  def init = { servletContext ->
    def jane = new Runner(
      firstName:"Jane",
      lastName:"Doe",
      dateOfBirth:(new Date() - 365*30),
      gender:"F",
      address:"123 Main St",
      city:"Goose",
      state:"NC",
      zipcode:"12345",
      email:"jane@whereever.com"
    )

    jane.save()
    if(jane.hasErrors()){
      println jane.errors
    }
  }

  def destroy = { }
}

```

Type `grails run-app` one more time, and this time you should see our good friend Jane at <http://localhost:9090/racetrack/runner/>:



Now that you know that it works for runners, let's add a race and a registration as well. Oh, and one more thing: I'll add in some logic to make sure that our dummy data (no offense, Jane) only appears in development mode.

```

import grails.util.GrailsUtil

class BootStrap {
  def init = { servletContext ->
    switch(GrailsUtil.environment){
      case "development":

        def jane = new Runner(
          firstName:"Jane",
          lastName:"Doe",
          dateOfBirth:(new Date() - 365*30),
          gender:"F",
          address:"123 Main St",
          city:"Goose",
          state:"NC",
          zipcode:"12345",
          email:"jane@whereever.com"
        )
        jane.save()
        if(jane.hasErrors()){
          println jane.errors
        }

        def trot = new Race(
          name:"Turkey Trot",
          startDate:(new Date() + 90),
          city:"Duck",
          state:"NC",
          distance:5.0,
          cost:20.0,
          maxRunners:350
        )
        trot.save()
        if(trot.hasErrors()){
          println trot.errors
        }

        def reg = new Registration(
          paid:false,
          runner:jane,
          race:trot
        )
        reg.save()
        if(reg.hasErrors()){
          println reg.errors
        }

        break

      case "production" : break
    }
  }
  def destroy = { }
}

```

With your `Bootstrap.groovy` file in place, type `grails run-app` once more to confirm that from this point forward you can focus on typing in code instead of repetitive sample data.

As we bring this chapter to a close, type `grails stats` to see where we stand:

```
$ grails stats
```

Name	Files	LOC
Controllers	3	9
Domain Classes	3	62
Unit Tests	6	68
Integration Tests	1	19
Totals	13	158

Things are really beginning to take shape, aren't they? And we're at just over 150 lines of code.

You now understand how to create a simple 1:M relationship with `hasMany`, `belongsTo`, and `toString()`. You learned that a M:M relationship is actually just two 1:M relationships with a third class that you haven't identified yet. And finally, you invested a little bit of time in `Bootstrap.groovy` so that you won't wear out your keyboard retyping data each time you reboot Grails.

### Free Online Version

Support this work, buy the print copy:

<http://www.infoq.com/minibooks/grails-getting-started>

# 6

## Databases

In the previous chapter, we used `Bootstrap.groovy` to keep our sample data around between Grails restarts. In this chapter, we'll explore the root causes of Grails' amnesia when running in Development mode. We'll also get Grails on speaking terms with a database other than HSQLDB.

### GORM

We've been creating domain classes willy-nilly, never too concerned about how or where the data is being persisted. GORM is to thank for our blissful ignorance. The Grails Object-Relational Mapping API allows us to stay squarely in an Object frame of mind – no getting bogged down in relational database-related SQL.

GORM is a thin Groovy façade over Hibernate. (I guess "Gibernate" doesn't roll off the tongue as nicely as GORM.) This means that any database that Hibernate supports, Grails (via GORM) does as well. This also means that all of your old Hibernate tricks can come along for the ride—if you have HBMs (Hibernate mapping files) or EJB3-annotated POJOs, you can use them unchanged. In this book, we'll focus on greenfield Grails development.

### DataSource.groovy

To better understand the default settings of GORM, open `grails-app/conf/DataSource.groovy` in a text editor.

```

dataSource {
    pooled = true
    driverClassName = "org.hsqldb.jdbcDriver"
    username = "sa"
    password = ""
}
hibernate {
    cache.use_second_level_cache = true
    cache.use_query_cache = true
    cache.provider_class =
        'com.opensymphony.oscache.hibernate.OSCacheProvider'
}
// environment specific settings
environments {
    development {
        dataSource {
            // one of 'create', 'create-drop', 'update'
            dbCreate = "create-drop"
            url = "jdbc:hsqldb:mem:devDB"
        }
    }
    test {
        dataSource {
            dbCreate = "update"
            url = "jdbc:hsqldb:mem:testDb"
        }
    }
    production {
        dataSource {
            dbCreate = "update"
            url = "jdbc:hsqldb:file:prodDb;shutdown=true"
        }
    }
}
}

```

Let's explore this file section by section.

The first block (labeled `dataSource`) contains the basic database settings that are shared across all environments. In this case, `development`, `test`, and `production` all share a common database driver, username, and password. (More on environments in just a moment.) If your production database requires a different set of credentials than your development database, you can set these values inside of the appropriate block. Anything in an environment block will override these “global” settings.

The `hibernate` block, as the name implies, is where you can adjust the default settings for Hibernate. There is no need to adjust these settings unless you are well versed in tuning Hibernate performance.

The last block (labeled `environments`) is where the interesting behavior happens. First of all, remember the “feature” that deletes your

data every time you reboot Grails? This happens because `dbCreate` is set to `create-drop`. The name pretty much says it all – every time you start Grails, the database tables are created. Every time you shut down Grails, the tables are dropped.

Notice that in `test` and `production`, `dbCreate` is set to `update`. In this mode, Hibernate leaves the tables and data intact and does a `SQL ALTER TABLE` to keep them in sync with the domain classes. (The third option – `create` – leaves the tables in place but deletes the data on shutdown.)

You might be tempted switch to `update` mode in `development`. While there is technically nothing wrong with doing so, bear in mind that the database schema will most likely be in an extreme state of flux during `development` – especially in the early stages. I’ve found that leaving it in `create-drop` mode and using `Bootstrap.groovy` is a better strategy for keeping everything in sync.

If the thought of letting GORM manage your database schema gives your DBAs the cold sweats, don’t worry. Commenting out this value tells GORM to do nothing at all. At this point you’ll be responsible for keeping the domain classes and database tables in sync yourself, but sometimes that can be the best strategy. If you are putting up a Grails application over a legacy, shared database, this is a perfect solution. (For Hibernate experts: the `dbCreate` variable maps directly to the `hibernate.hbm2ddl.auto` value.)

The other setting – `url` – is the JDBC connection string. Notice that `production` mode uses a file-based `HSQLDB`, while `test` and `development` modes both use a memory-based one.

## Switching to an External Database

`HSQLDB` is a great database for getting up and running in a hurry, but you’ll most likely want to switch to an external database before you go into `production`. To do so, you’ll need to do three simple things:

1. Create the database, along with a login and password
2. Copy the JDBC driver to the `grails-app/lib` directory
3. Adjust the settings in `grails-app/conf/DataSource.groovy`

To start, I’ll show you how to create a new database in `MySQL`. (For other databases, read the fine manual that accompanies them.) If you

didn't install MySQL earlier, now is a good time to do so. I'll wait right here.

OK, to create a new database, you'll need to log in as the root user. You can probably get by with just creating a database for development mode right now, although if you're feeling ambitious, feel free to create databases for test and production mode as well.

```
$ mysql -u root -p
Enter password: *****
Welcome to the MySQL monitor.  Commands end with ; or \g.
...
mysql> create database racetrack_dev;
...
mysql> grant all on racetrack_dev.* to grails@localhost
identified by 'server';
...
mysql> exit
```

The next to last command probably deserves a bit of explanation. You are granting all privileges (the ability to create tables, drop tables, alter tables, etc.) for all objects in the `racetrack_dev` database to a user named `grails`. This user can only log in from `localhost` – you could change `localhost` to an IP address or another host name if the database is on a different server than the Grails application. Finally, in this example, that user's password is `server`.

To verify that everything is configured correctly on the MySQL end, log back into MySQL using your new credentials.

```
$ mysql --user=grails -p --database=racetrack_dev
Welcome to the MySQL monitor.

Mysql> show tables;
Empty set (0.00 sec)
```

OK, so you've confirmed that you can log in to MySQL and that the database is empty.

The next step is to copy the MySQL driver into the `grails-app/lib` directory. You can download the latest Java MySQL driver from <http://dev.mysql.com/downloads/connector/j/>. Unzip the file and look for the JAR file. (It should be named something like `mysql-connector-java-5.1.5-bin.jar`.)

Finally, adjust `grails-app/conf/DataSource.groovy` to your new settings. (Note: each URL should be on a single line in your file.)



```

dataSource {
    pooled = true
    driverClassName = "com.mysql.jdbc.Driver"
    username = "grails"
    password = "server"
}
hibernate {
    cache.use_second_level_cache = true
    cache.use_query_cache = true
    cache.provider_class =
        'com.opensymphony.oscache.hibernate.OSCacheProvider'
}
// environment specific settings
environments {
    development {
        dataSource {
            // one of 'create', 'create-drop', 'update'
            dbCreate = "create-drop"
            // NOTE: the JDBC connection string should be
            //      all on the same line.
            url = "jdbc:mysql://localhost:3306/racetrack_dev?
                autoreconnect=true"
        }
    }
    test {
        dataSource {
            dbCreate = "update"
            url = "jdbc:mysql://localhost:3306/racetrack_dev?
                autoreconnect=true"
        }
    }
    production {
        dataSource {
            dbCreate = "update"
            url = "jdbc:mysql://localhost:3306/racetrack_dev?
                autoreconnect=true"
        }
    }
}

```

Did you notice that I pointed both test and production modes to the same database as development? That's just me being lazy – I don't recommend doing that for a real production application. If you created separate MySQL databases for each mode (and you should!), you can make the appropriate changes in the appropriate spots.

Now is the moment of truth: did we really get everything configured correctly for Grails to use MySQL? There is only one way to find out. Type `grails run-app` and look for errors on the console. If everything looks OK there, the next place to check is in MySQL.

```
$ mysql --user=grails -p --database=racetrack_dev
Welcome to the MySQL monitor.
```

```
mysql> show tables;
```

```
+-----+
| Tables_in_racetrack_dev |
+-----+
| race                     |
| registration             |
| runner                   |
+-----+
3 rows in set (0.01 sec)
```

```
mysql> desc registration;
```

```
+-----+-----+-----+-----+
| Field      | Type      | Null | Key |
+-----+-----+-----+-----+
| id          | bigint(20)| NO   | PRI |
| version     | bigint(20)| NO   |     |
| date_created | datetime  | NO   |     |
| paid        | bit(1)    | NO   |     |
| race_id     | bigint(20)| NO   | MUL |
| runner_id   | bigint(20)| NO   | MUL |
+-----+-----+-----+-----+
6 rows in set (0.02 sec)
```

```
mysql> desc race;
```

```
+-----+-----+-----+-----+
| Field      | Type      | Null | Key |
+-----+-----+-----+-----+
| id          | bigint(20)| NO   | PRI |
| version     | bigint(20)| NO   |     |
| city        | varchar(255)| NO   |     |
| cost        | decimal(5,2)| NO   |     |
| distance    | decimal(19,2)| NO   |     |
| max_runners | int(11)    | NO   |     |
| name        | varchar(50)| NO   |     |
| start_date  | datetime   | NO   |     |
| state       | varchar(2) | NO   |     |
+-----+-----+-----+-----+
9 rows in set (0.01 sec)
```

```
mysql> desc runner;
```

```
+-----+-----+-----+-----+
| Field      | Type      | Null | Key |
+-----+-----+-----+-----+
| id          | bigint(20)| NO   | PRI |
| version     | bigint(20)| NO   |     |
| address     | varchar(255)| NO   |     |
| city        | varchar(255)| NO   |     |
| date_of_birth | datetime  | NO   |     |
| email       | varchar(255)| NO   |     |
| first_name  | varchar(255)| NO   |     |
| gender      | varchar(1) | NO   |     |
| last_name   | varchar(255)| NO   |     |
| state       | varchar(255)| NO   |     |
| zipcode     | varchar(255)| NO   |     |
+-----+-----+-----+-----+
11 rows in set (0.01 sec)
```

Grails seems to be successfully talking to MySQL at this point. Now that wasn't too tough at all, was it?

Notice that every class in the `grails-app/domain` directory has a corresponding table. Every attribute has a corresponding column. Camel-case fields names (like `dateOfBirth`) are converted to all lower case names with underscores (like `date_of_birth`).

There are two additional fields that GORM adds to every class – `id` and `version`. The `id` is, of course, the primary key. This is what allows the foreign keys in the `registration` table to work.

The `version` field holds a simple integer that gets incremented each time the record gets updated. This is called *optimistic locking* in Hibernate parlance. If two users try to edit the same record at the same time, the first update increments the `version` field. When the second user tries to save the record, GORM can easily see that the record is out of date and will warn the user accordingly.

In this chapter, you finally learned why Grails seems to forget your data between restarts. To fix this, you simply adjust the `dbCreate` value in `grails-app/conf/DataSource.groovy` from `create-drop` to `update`. (If you are using HSQLDB, you also need to switch your development environment from an in-memory database to a file-backed database. One way to accomplish this task is to copy the `url` from production to development.) You also learned how to point Grails to an external database.

Now that we've solved the database configuration puzzle, let's turn our focus to the next major piece: Controllers.



## Free Online Version

Support this work, buy the print copy:

<http://www.infoq.com/minibooks/grails-getting-started>

# 7

## Controllers

In this chapter, we'll expose the default scaffolded Controller code so that we can make tiny adjustments to it. This will help us better visualize the relationship between closures (or actions) in the Controllers and the URL that the end-user sees. It will also allow us to see the relationship between closures and the corresponding Groovy Server Pages (GSPs).

### create-controller vs. generate-controller

Up until now, we've been relying on the magic `def scaffold = true` statement in our Controllers to supply the full CRUD interface. This statement not only supplies the behavior for the Controller; it also generates the GSPs for the view. For the next class we create, let's expose the underpinnings so that we can get a better idea of what is really going on behind the scenes.

Let's create a new `User` domain class. (We'll use this class in the upcoming *Security* chapter.) Type `grails create-domain-class User` and add the following code:

```
class User {
    String login
    String password
    String role = "user"

    static constraints = {
        login(blank:false, nullable:false, unique:true)
        password(blank:false, password:true)
        role(inList: ["admin", "user"])
    }

    String toString(){
        login
    }
}
```

Normally I'd have you type `grails create-controller User` at this point. Recall that the `create-*` commands create an empty, stubbed out class. Instead, type `grails generate-all User`. The `generate-all` command writes out a fully implemented controller and a corresponding set of GSP views. (You can also type `generate-views` to see just the GSPs, or `generate-controller` to create just the Controller.)

Instead of the one-line Controller you are used to seeing, you'll now see 100 lines of Groovy code (edited here for brevity):

```
class UserController {

    static allowedMethods = [save: "POST",
                           update: "POST",
                           delete: "POST"]

    def index = {
        redirect(action: "list", params: params)
    }

    def list = {
        params.max = Math.min(params.max ?
                               params.int('max') : 10, 100)
        [userInstanceList: User.list(params),
         userInstanceTotal: User.count()]
    }

    def create = {
        def userInstance = new User()
        userInstance.properties = params
        return [userInstance: userInstance]
    }

    def save = { // ... }
    def show = { // ... }
    def edit = { // ... }
    def update = { // ... }
    def delete = { // ... }
}
```

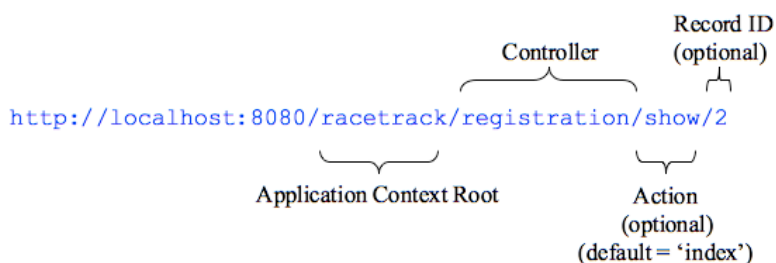
Now that we've exposed the implementation details, let's see how all of the pieces fit together.

## Understanding URLs and Controllers

Consider the types of URLs we saw as we worked our way through the application:

<http://localhost:9090/racetrack/registration>  
<http://localhost:9090/racetrack/registration/create>  
<http://localhost:9090/racetrack/registration/show/2>

Each component of the URL plays an important role in the Grails convention. And it's by following that convention that we (thankfully) free ourselves from having to wire together our URLs, controllers, and views in external configuration files.



**Figure 3-1: URL Structure**

The first part of the URL is the application name, or Application Context Root. Not surprisingly, this is the name of the directory that was created when we typed `grails create-app racetrack`. The application name and the directory name, however, don't have to match. If you open `application.properties` in the root folder of the RaceTrack application, you'll see that you can set the `app.name` to any arbitrary value you'd like. The new value will then appear as the first part of the URL.

```
app.version=0.1
app.name=racetrack
app.servlet.version=2.4
app.grails.version=1.2
plugins.hibernate=1.2
plugins.tomcat=1.2
```

The next part of the URL is the controller name. Notice that the Controller part of `RegistrationController` gets dropped off, and the first letter is changed to lowercase.

After the controller name, you'll see an optional action name. This name corresponds directly to the closures in the Controller. If you have a closure named `list`, the URL is `registration/list`. If you leave the action off, the default `index` closure gets called.

Looking back at the generated `UserController`, notice that the `index` closure does a simple redirect to the `list` action. The second argument – `params` – is a `HashMap` of the name / value pairs on the query string. If you type in an URL like `http://localhost:9090/racetrack/registration/list?max=3&sort=paid`, you can access the values from the query string in your Controller using `params.max` and `params.sort` respectively.

The last part of the URL, also optional, is the spot where the primary key usually appears. When you request `edit/2`, `show/2`, or `delete/2`, you are asking for the record with a primary key of 2 to be edited, shown, or deleted. Since dealing with record IDs is so common, Grails automatically assigns this value to `params.id` for convenience.

Let's see if we can trace the workflow of a typical web request from start to finish.

## From Request to Controller to View

Here is a snippet from the `UserController` you created just a moment ago.

```
class UserController {
    def index = {
        redirect(action: "list", params: params)
    }

    def list = {
        params.max = Math.min(params.max ?
                               params.int('max') : 10, 100)
        [userInstanceList: User.list(params),
         userInstanceTotal: User.count()]
    }
}
```



Type <http://localhost:9090/racetrack/user> in your web browser. You should end up with an empty list of users. So, how did you get there?

First, because no action is specified, Grails routes the request to the `index` action. The `index` action, in turn, redirects the request to the `list` action, passing along the query string as well. The `name:value` structure used in the `redirect` method is used throughout Controllers. In fact, it is nothing more than a simple `HashMap` of key/value pairs.

*Three R's:* Action closures typically end in one of three ways, each beginning with the letter R. (I affectionately call this the “Three R’s of Controllers”.)

The first R – `redirect` – bounces the request from one action to another. The second R – `render` – sends arbitrary text back to the browser. (Think back to the earlier, goofy `render` “Hello World” examples.) The `render` command can render more than simple text, as you’ll see later. It can render XML, JSON, HTML, and much more. The final R is a `return`. In some cases, `return` is typed in explicitly. Other times, as we’ll see in the case of `list`, the `return` is implicit.

So, we’ve now moved from the `index` action to the `list` action. The first line of `list` sets the `max` parameter if it isn’t passed in via the query string. This value is used for pagination. (Once you get more than 10 Runners, Races, or Registrations, Grails will create a second page with navigation links. To force pagination for fewer than 10, pass in the `max` parameter, like `runner/list?max=2`.)

The last line of `list` is the `return` statement. How can you tell? Well, the last line of any Groovy method is an implicit `return` statement. In this case, the `list` action is returning a `HashMap` of values to the GSP view. (More on that in just a moment.)

The first item in the `HashMap` is a list of `Users`, limited by the `max` parameter. The next item is the total number of users, used once again for pagination purposes.

But where did these static methods come from? Looking at `User.groovy`, I don’t see a static `list()` or `count()` method anywhere. These methods are added to the `User` class dynamically at runtime by GORM. (This is what the cool kids call *metaprogramming*.)

Scan the rest of the `UserController` source code, and you'll see tons of methods added to the `User` class – `save()`, `delete()`, `get()`, and so on.

So, we now know that the last line of the `list` action is a `return` statement. But one question remains: how do we go from a `HashMap` containing a list of `Users` to an HTML table displaying them? In other words, where does this `return` statement take us next?

Well, it's convention (once again) that allows the `list` action to be so concise. Because we're inside `UserController` and returning from the `list` action, Grails knows to use the view template located at `grails-app/views/user/list.gsp`.

## A Quick Look at GSPs

---

We'll spend more time in the next chapter exploring GSPs in depth. But just to complete our walk-through of the request lifecycle, open `grails-app/views/user/list.gsp` in a text editor. Here's a snippet from the body:

```
<g:each in="{userInstanceList}"
    status="i"
    var="userInstance">
  <tr class="{(i % 2) == 0 ? 'odd' : 'even'}">
    <td>
      <g:link action="show" id="{userInstance.id}">
        ${fieldValue(bean:userInstance, field:'id')}
      </g:link>
    </td>
    <td>${fieldValue(bean:userInstance,
      field:'login')}</td>
    <td>${fieldValue(bean:userInstance,
      field:'password')}</td>
    <td>${fieldValue(bean:userInstance,
      field:'role')}</td>
  </tr>
</g:each>
```

Notice the `userInstanceList`? That's the same name returned from the `list` action. Whatever the Controller returns, the GSP can use directly.

Later in `list.gsp`, you can see the pagination code in action:

```
<div class="paginateButtons">
  <g:paginate total="${userInstanceTotal}" />
</div>
```

The `userInstanceTotal` value returned from the Controller is used in the custom `g:paginate` tag.

## Exploring the Rest of the Controller Actions

Now that you have the full request lifecycle under your belt, feel free to browse the rest of the Controller actions on your own. In each case, the action is extracted from the URL, passed to the appropriate closure, and then ultimately passed on to a GSP.

When the URL is `user/show/2`, the first place to look is in `UserController` for a `show` closure.

```
def show = {
  def userInstance = User.get(params.id)
  if (!userInstance) {
    flash.message = "${message(code: 'default.not.found.message',
      args: [message(code: 'user.label',
        default: 'User'), params.id])}"
    redirect(action: "list")
  }
  else {
    [userInstance: userInstance]
  }
}
```

GORM attempts to get the `User` with a primary key of “2” (stored in `params.id`) from the database. If that `User` can't be found, a message is placed in flash scope (like request scope, only slightly longer lived so that it will survive the `redirect`). If, on the other hand, the `User` is found, a `HashMap` with the `User` in it is returned.

*Scopes in Grails:* There are four scopes in Grails: request, flash, session, and application. Each one has a different lifespan.

- Variables placed in request scope last just long enough to be displayed in the response and then are discarded.
- Values in flash scope (commonly used for error messages in Grails) can survive a single redirect.
- Session scope is used for values that should stick around for multiple requests, but are limited to the current user.
- Values placed in application scope are “global”—they last for as long as Grails is running, and they are shared across all users.

Where does the `HashMap` go? Why, to `grails-app/views/user/show.gsp`, of course. Throughout `show.gsp`, you’ll see `userInstance` in action:

```
<tr class="prop">
  <td valign="top" class="name">
    <g:message code="user.id.label" default="Id" />
  </td>

  <td valign="top" class="value">
    ${fieldValue(bean: userInstance, field: "id")}
  </td>
</tr>

<tr class="prop">
  <td valign="top" class="name">
    <g:message code="user.login.label" default="Login" />
  </td>

  <td valign="top" class="value">
    ${fieldValue(bean: userInstance, field: "login")}
  </td>
</tr>
```

Easy, right? The `show` URL leads to the `show` action, which in turn leads to the `show.gsp` view. Convention over configuration at its finest!

## Rendering Views That Don't Match Action Names

While conventions are powerful, sometimes you need to break the rules. There are times when you want to use a view that doesn't match the name of the action.

For example, when you create a new `User`, the form gets submitted to the `save` action. Look at the form action in `create.gsp` to verify this.

```
<g:form action="save" method="post" >
  <!-- ... -->
</g:form>
```

But there is no corresponding `save.gsp` in the `grails-app/views/user` directory. How does `save` deal with this?

```
def save = {
  def userInstance = new User(params)
  if (userInstance.save(flush: true)) {
    flash.message = "${message(code: 'default.created.message',
                               args: [message(code: 'user.label',
                                                default: 'User'), userInstance.id])}"
    redirect(action: "show", id: userInstance.id)
  }
  else {
    render(view: "create", model: [userInstance: userInstance])
  }
}
```

If the `User` is saved successfully, the `save` action simply redirects to the `show` action. But if there is a problem saving the `User` (most likely due to a validation problem), the `save` action renders the `create.gsp` view directly.

In this chapter, we took a deeper dive into Controllers. We generated a Controller instead of scaffolding it, thereby exposing its internal workings. This allows us to see that each closure (or action) in the Controller shows up in the URL. We also learned that actions can return a `HashMap` of values to a GSP of the same name, redirect to another action, or render an arbitrary String or GSP of any name.

In the next chapter, we'll take a deeper dive into GSPs.



## Free Online Version

Support this work, buy the print copy:

<http://www.infoq.com/minibooks/grails-getting-started>

# 8

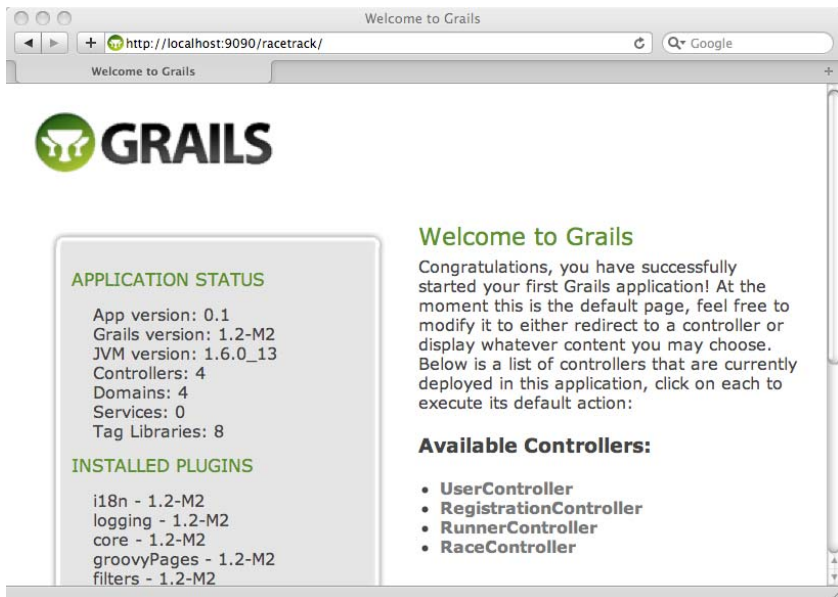
## Groovy Server Pages

In this chapter, we'll learn the ins and outs of generating the views of a Grails application. You'll see how SiteMesh interleaves your GSPs. You'll create partial templates, custom taglibs, and much more.

Things move pretty fast in this chapter. If you already are familiar with HTML and basic web development, you shouldn't have any trouble keeping up. If you need to brush up on your client-side web skills, see <http://htmlcodetutorial.com>.

## Understanding GSPs

To begin, type `grails run-app` and visit the home page in a web browser.



Hmm – I don't see a controller on the URL, so we must be hitting a GSP directly. Since index is the default action in a controller, I wonder if there is an index.gsp page somewhere? Sure enough, in grails-app/views there is an index.gsp file. Open it in a text editor.

```
<html>
  <head>
    <title>Welcome to Grails</title>
    <meta name="layout" content="main" />
    <!-- CSS styling snipped -->
  </head>
  <body>
    <div id="nav">
      <div class="homePagePanel">
        <div class="panelTop"></div>
        <div class="panelBody">
          <h1>Application Status</h1>
          <ul>
            <li>App version:
              <g:meta name="app.version"></g:meta>
            </li>
            <li>Grails version:
              <g:meta name="app.grails.version"></g:meta>
            </li>
            <li>JVM version:
              ${System.getProperty('java.version')}
            </li>
            <li>Controllers:
              ${grailsApplication.controllerClasses.size()}
            </li>
            <li>Domains:
              ${grailsApplication.domainClasses.size()}
            </li>
            <li>Services:
              ${grailsApplication.serviceClasses.size()}
            </li>
            <li>Tag Libraries:
              ${grailsApplication.tagLibClasses.size()}
            </li>
          </ul>
          <h1>Installed Plugins</h1>
          <ul>
            <g:set var="pluginManager"
value="${applicationContext.getBean('pluginManager')}">
              </g:set>
              <g:each var="plugin"
in="${pluginManager.allPlugins}">
                <li>${plugin.name} - ${plugin.version}</li>
              </g:each>
            </ul>
          </div>
          <div class="panelBtm"></div>
        </div>
      </div>
    </div>
  </div>
```



```

<div id="pageBody">
  <h1>Welcome to Grails</h1>
  <p>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:
  </p>

  <div id="controllerList" class="dialog">
    <h2>Available Controllers:</h2>
    <ul>
      <g:each var="c"
        in="${grailsApplication.controllerClasses}">
        <li class="controller">
          <g:link
            controller="${c.logicalPropertyName}">
            ${c.fullName}
          </g:link>
        </li>
      </g:each>
    </ul>
  </div>
</body>
</html>

```

It seems to be pretty much a basic HTML file. There are, however, some `<g:>` elements that you might not recognize. These are Grails tags. Throughout this chapter, you'll not only learn many of the default Grails tags, but also end up creating some of your own. But the important lesson here is that a GSP is simply straight HTML + Grails tags.

A common Grails tag you'll see used over and over again is `<g:each>`. Seeing how it is used here (as well as in the `list.gsp` snippet in the previous chapter) probably gives you a pretty good idea of what it does – it iterates over each item in a collection. In this case, it displays each controller in our Grails application.

You'll also see `<g:link>` frequently used. It creates – wait for it – a link to the named controller. (This really isn't tough stuff, is it?) If you view the source of the page in your web browser, you'll see a series of hyperlinks that contain well-formed URLs:

```

<ul>
  <li class="controller">
    <a href="/racetrack/runner">RunnerController</a>
  </li>

  <li class="controller">
    <a href="/racetrack/race">RaceController</a>
  </li>

  <li class="controller">
    <a href="/racetrack/registration">
      RegistrationController</a>
  </li>

  <li class="controller">
    <a href="/racetrack/user/index">UserController</a>
  </li>
</ul>

```

If you want to create a link to a specific action (the ones in `index.gsp` will, as you learned in the last chapter, take the user to the default index action), simply add an `action` attribute:

```

<g:link controller="user" action="create">
  New User
</g:link>

```

There are far too many Grails tags to cover in-depth here. I think the best way to learn about them is to see them in action in the generated GSPs. For example, at the top of every `list.gsp` file, you'll see this:

```

<g:if test="${flash.message}">
  <div class="message">${flash.message}</div>
</g:if>

```

As you probably guessed, this snippet checks to see if there is a message value stored in flash scope. If there is, it displays the `<div>` enclosed in the `<g:if>` tag.

For a more detailed listing of all of the available Grails tags, see the excellent online documentation: <http://grails.org/Documentation>.

## Understanding SiteMesh

One thing in `index.gsp` is a little odd – I can't see where the Grails logo is coming from, can you? I can see the link when I view the source from my web browser, but I can't see it anywhere in `index.gsp`. What do you think is going on?

Grails uses a popular templating library called SiteMesh. SiteMesh quite literally *meshes* two GSPs together. It's one way that Grails factors out common behavior into a reusable place. The Grails logo, the shared CSS file, and much more is stored in a single file that is combined with the `index.gsp` file you have been looking at.

Notice the `<meta>` tag in the `<head>` section?

```
<head>
  <title>Welcome to Grails</title>
  <meta name="layout" content="main" />
</head>
```

This subtle hint (some might argue *too* subtle!) tells you that there is a template called `main.gsp` in the `grails-app/views/layouts` directory. Open it in a text editor.

```
<html>
  <head>
    <title><g:layoutTitle default="Grails" /></title>
    <link rel="stylesheet"
      href="{resource(dir:'css',file:'main.css')}" />
    <link rel="shortcut icon"
      href="{resource(dir:'images',file:'favicon.ico')}"
      type="image/x-icon" />
    <g:layoutHead />
    <g:javascript library="application" />
  </head>
  <body>
    <div id="spinner" class="spinner"
      style="display:none;">
      
    </div>
    <div id="grailsLogo" class="logo">
      <a href="http://grails.org">
        
      </a>
    </div>
    <g:layoutBody />
  </body>
</html>
```

Based on the `<g:layoutHead>` and `<g:layoutBody>` tags, it looks like this file is merged with the `index.gsp` file we saw earlier. In fact, that is exactly what is going on. This file's `<head>` section is meshed with the other GSP file's `<head>` section. The same is true with the two `<body>` sections.

To test this theory, comment out the `<div>` that displays the Grails logo, save the file, and refresh the view in your web browser.

```
<!-- bye bye, logo
<div id="grailsLogo" class="logo">
  <a href="http://grails.org">
    
  </a>
</div>
-->
```

Did the logo disappear? OK, so now we're getting somewhere. (By the way, did you notice that the `resource` tag creates a link to a file in a directory, while `link` creates a link to an action in a controller? Did you also notice that you can call taglibs either using `<g:>` syntax or `${ }` syntax?)

Next, let's create a custom header for our application. We could begin adding code directly into the SiteMesh template, but I'd like to introduce you to another way that you can factor out common artifacts in GSPs.

SiteMesh is great for meshing two entire GSPs, but what if you want to do things at a smaller scale? What if you just want to have a partial snippet of GSP code? Partial templates are the answer.

## Understanding Partial Templates

To create a partial template in Grails, simply prefix the GSP filename with an underscore. (As you are scanning files in the `views` directory, the leading underscore provides a quick visual cue that the file isn't a full, well-formed view.) For example, create a file named `_header.gsp` in `grails-app/views/layouts`. Add the following code to it:

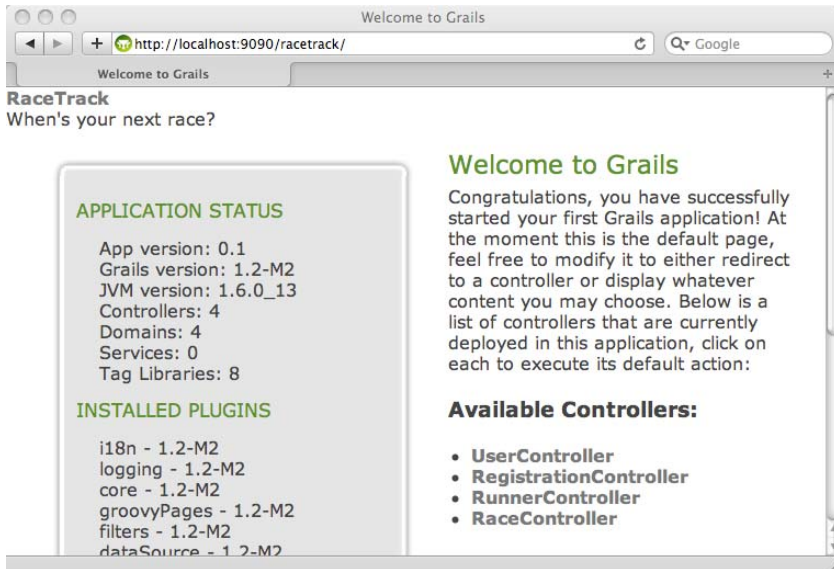
```
<div id="header">
  <p><a class="header-main"
    href="${resource(dir: '')}">RaceTrack</a></p>
  <p class="header-sub">When's your next race?</p>
</div>
```

Now let's put it in action. Go back to `main.gsp` and render the partial template *inside* `main.gsp` using the `<g:render>` tag:

```
<body>
  <!-- snip -->
  <g:render template="/layouts/header" />
  <g:layoutBody />
</body>
```

The root directory, in this context, is the `views` directory. Since we placed the partial template in `grails-app/views/layouts`, the path to the template is `/layouts/`. And yes, you leave both the leading underscore and the trailing `.gsp` off when using the `<g:render>` tag. (This can be a bit confusing the first time you see it, but after a while you won't think twice about it.)

Refresh your browser once again. Your screen should look something like this:



Now that the header partial template is in place, let's add a bit of CSS formatting to make it look better. Add this code to the bottom of `web-app/css/main.css`.

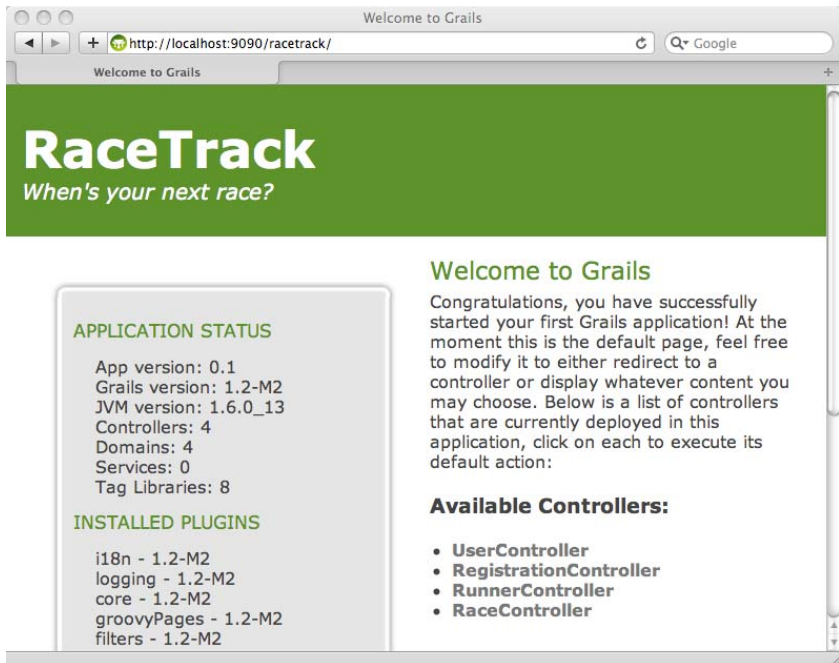
```
/* RaceTrack Customization */
#header {
    background: #48802c;
    padding: 2em 1em 2em 1em;
    margin-bottom: 1em;
}
a.header-main:link, a.header-main:visited {
    color: #fff;
    font-size: 3em;
    font-weight: bold;
}
.header-sub {
    color: #fff;
    font-size: 1.25em;
    font-style: italic;
}
```

*Quick Note:* CSS entries that start with # refer to IDs (like header). CSS entries that start with . (a dot) refer to classes (like header-main and header-sub). To brush up on your CSS skills, see <http://w3.org/Style/Examples/011/firstcss>.

Did you notice the line in the SiteMesh template in `grails-app/views/layouts/main.gsp` that includes this CSS file?

```
<head>
  <link rel="stylesheet"
        href="${resource(dir:'css',
                           file:'main.css')}" />
  <g:layoutHead />
</head>
```

Refresh your browser again to see the results:



Wow – that’s really beginning to look nice. Let’s pause for a moment and recap what we’ve accomplished thus far. You now know that Grails uses SiteMesh to merge common, shared elements with individual GSPs using the `<g:layoutHead>` and `<g:layoutBody>` plug-points. While that’s nice for merging two entire pages together, partial templates allow you to do the same thing at a smaller scale. You can include tiny snippets of GSP code from a partial template using the `<g:render>` tag. Make sense?

Since you already know how to create a partial template for the header, let's create a partial template for the footer as well. Create `grails-app/views/layouts/_footer.gsp` with the following content.

```
<div id="footer">
  <hr />
  &copy; 2009 Racetrack, Inc.
</div>
```

Now add the template to `main.gsp`:

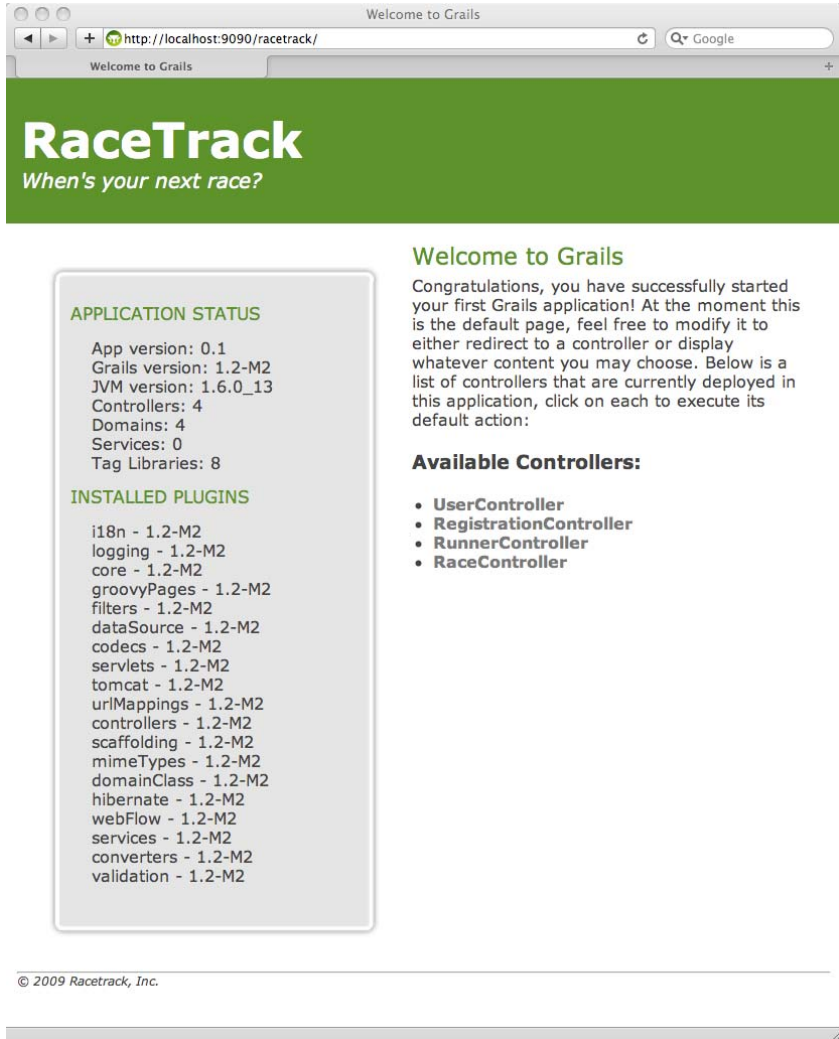
```
<body>
  <g:render template="/layouts/header" />
  <g:layoutBody />
  <g:render template="/layouts/footer" />
</body>
```

And finally, add a bit of CSS to `main.css`:

```
#footer {
  font-size: 0.75em;
  font-style: italic;
  padding: 2em 1em 2em 1em;
  margin-bottom: 1em;
  margin-top: 1em;
  clear: both;
}
```



Refresh your browser to see the results:



Everything looks good, except now something else is bugging me. I don't like the fact that the year is hard-coded into the footer template. Wouldn't it be nice if there were something smaller than a partial template? Something that could return a simple value? Creating your own custom TagLib is just what you are looking for.

## Understanding Custom TagLibs

Since Grails provides a number of `<g:>` tags already, it shouldn't surprise you that you can create your own tags as well. What might surprise you is how easy it is.

Type `grails create-tag-lib Footer` at the command prompt. Just like `create-controller` and `create-domain-class`, this creates an empty, stubbed-out `FooterTagLib.groovy` in the `grails-app/taglib` directory. It also creates a corresponding test class. Open `FooterTagLib.groovy` in a text editor.

```
class FooterTagLib {

}
```

Not much to look at right now, is there? Now add the following code:

```
class FooterTagLib {

    def thisYear = {
        out << new Date().format("yyyy")
    }

}
```

That's all it takes to create a new `<g:thisYear />` tag. `out` is the output stream, and you are sending the current year to it. The `format()` method on `Date` is identical to the one found in `java.text.SimpleDateFormat`. Groovy *metaprograms* this new method onto `java.util.Date` as a convenience, much like GORM metaprograms `list()`, `save()`, and `delete()` methods onto your domain classes.

To give it a try, edit `grails-app/views/layouts/_footer.gsp` to take advantage of your new tag:

```
<div id="footer">
    <hr />
    &copy; <g:thisYear /> Racetrack, Inc.
</div>
```

Type `grails run-app` and refresh your browser to verify that nothing has changed. (All that work for nothing? OK, wait until 11:59 PM on December 31<sup>st</sup> and furiously click refresh for 60 seconds. Or you could sleep well that night knowing that you won't have to edit your Grails application with your annual New Year's Day hangover.)

Creating a simple custom tag is a quick and easy task, but what if you want to put together a more complex taglib? Say, for example, that you wanted to create a `<g:copyright>` tag that would output the copyright date as a range (e.g., 1999 – 2009) instead of just a single year. You might want to explicitly specify the start year (e.g., 1999), and have the tag dynamically use the current year as the end year. The finished tag will look like this:

```
<div id="footer">
  <hr />
  <g:copyright
    startYear="1999">Racetrack, Inc.</g:copyright>
</div>
```

To create this more advanced custom tag, add a copyright closure to `FooterTagLib.groovy` like so:

```
class FooterTagLib {

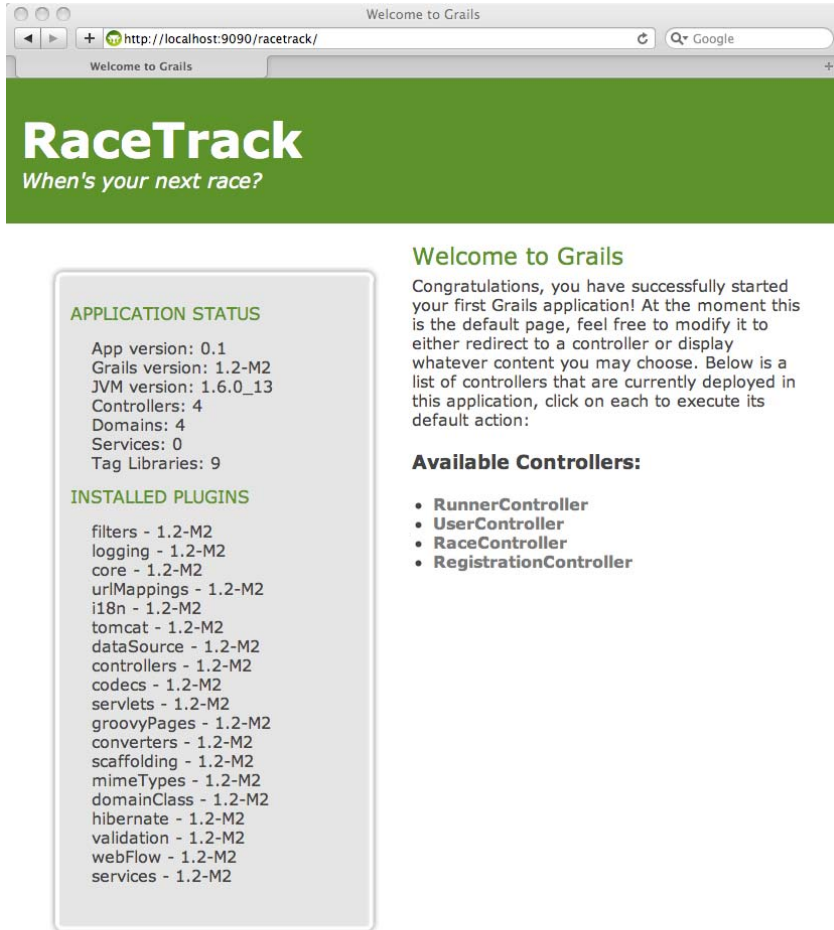
  def thisYear = {
    out << new Date().format("yyyy")
  }

  def copyright = {attrs, body->
    out << "&copy; " + attrs.startYear + " - "
    out << thisYear() + " " + body()
  }

}
```

Notice the two parameters in the closure declaration? `attrs` is a `HashMap` of attributes. Just like `params` and `flash` in a Controller, `attrs` gives you access to the `startYear` parameter on the `<g:copyright>` tag. The `body` tag, on the other hand, is passed as a closure that you need to call like a method. That's why you see `body()` (with parentheses) instead of simply `body`. Did you notice that you can invoke a tag (like `thisYear()`) as a method inside other tags as well?

Refresh your browser to see the results:

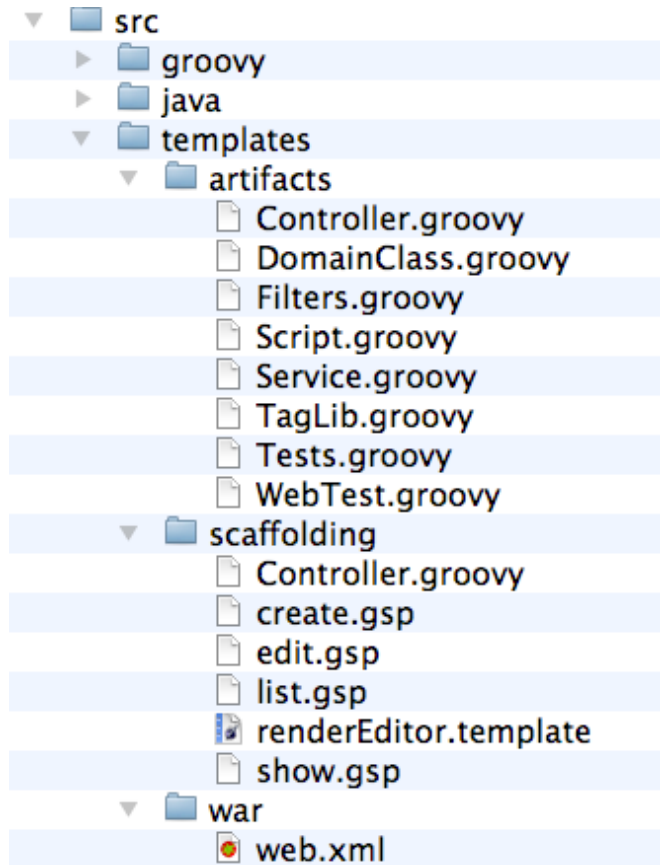


So, we're getting pretty good at customizing content around the edges of the scaffolded views. But what about customizing the inside of the scaffolded views? Can we customize Grails itself?

## Customizing the Default Templates

The magic of `def scaffold = true` is only really powerful if you can customize it as well. Thankfully, we can ask Grails to expose its inner workings so that we can deeply customize it down to the bare metal. The difference is this time, rather than customizing one-off code on a per-domain class basis, I will show you how to customize *all scaffolded views at once*.

To truly get at the inner workings of Grails, type `grails install-templates`. This command instructs Grails to make the template code for all views, controllers, domain classes, taglibs, etc. available for tweaking. Look in `src/templates` for boilerplate code for all of these files and more.



OK, so what can you do with these boilerplate files? Let's say that you want all controllers to be scaffolded out by default. Now that you have the templates installed, you can simply tweak `src/templates/artifacts/Controller.groovy` to look like this:

```
@artifact.package@class @artifact.name@ {
    def scaffold = true
}
```

Every new controller that you create by typing `grails create-controller` will use this template as its source.

As another example, suppose that you want to stub out a `toString()` method for all of your domain classes. You can make the adjustment in `src/templates/artifacts/DomainClass.groovy` as follows:

```
@artifact.package@class @artifact.name@ {

    static constraints = {
    }

    String toString(){
        // TODO: make me interesting
    }
}
```

As you can see, customizing these artifacts early in the development process can save you from needlessly copying and pasting snippets of code each time you create a new domain class, controller, etc.

While the `artifacts` directory supplies the templates for the `grails create-*` commands, the `scaffolding` directory is the source of the interesting GSP bits. What are some useful things that you can do to the scaffolded GSP files?

First of all, did you notice that the magic `dateCreated` and `lastUpdated` fields appear when you scaffold out the views? (As a reminder, we added a `dateCreated` field to the `Registration` class. Recall that fields with these names are autopopulated by Grails behind the scenes.)

Create Registration

Home Registration List

## Create Registration

Race: Turkey Trot, 11/02/2009

Runner: Doe, Jane (jane@whereever.com)

Paid: ☐

Date Created: 2 September 2009 22:00

Create

That's kind of a bummer, isn't it? After all, those timestamps can't really be adjusted by the user anyway. Wouldn't it be nice if you could exclude these properties from the scaffolded views? Thankfully, now that you have the templates installed, you can take matters into your own hands.

Look for the `excludedProps` list in `src/templates/scaffolding/create.gsp` and `edit.gsp`. As the name implies, any domain class fields (or properties) that appear in this list will be excluded from the scaffolded views. Add `dateCreated` and `lastUpdated` to the list:

```
<div class="dialog">
  <table>
    <tbody>
      <%
        excludedProps = Event.allEvents.toList()
        excludedProps << 'version'
        excludedProps << 'id'
        excludedProps << 'dateCreated'
        excludedProps << 'lastUpdated'
        props = domainClass.properties.findAll {
          !excludedProps.contains(it.name) }
      %>
    </tbody>
  </table>
</div>
```

Start up Grails once again, visit <http://localhost:9090/racetrack/registration/create>, and voilà – no more timestamps.

The screenshot shows a web browser window titled "Create Registration" with the URL <http://localhost:9090/racetrack/registration/create>. The page has a green header with the "RaceTrack" logo and the tagline "When's your next race?". Below the header is a navigation bar with "Home" and "Registration List" links. The main content area is titled "Create Registration" and contains a form with the following fields:

- Race:** A dropdown menu showing "Turkey Trot, 12/01/2009".
- Runner:** A text input field containing "Doe, Jane (jane@whereever.com)".
- Paid:** A checkbox that is currently unchecked.

At the bottom of the form is a "Create" button. Below the form, there is a footer line that reads "© 1999 - 2009 Racetrack, Inc.".

As another example – one that will lead us right into the discussion of authorization and authentication in the next chapter – suppose that you'd like to hide the navigation bar in the list view if the user isn't logged in as an administrator. Remember earlier in the chapter when you saw the `<g:if>` test that only displayed the `flash.message` if the message existed? You can use the same technique here.

Open `src/templates/scaffolding/list.gsp` in a text editor. Look for the `nav` div right after the `body` tag. If you wrap it in a simple `<g:if>` test, you can hide it in one place instead of repeatedly generating and editing each `list.gsp` by hand.



```

<g:if test="\${session?.user?.admin}">
  <div class="nav">
    <span class="menuButton">
      <a class="home" href="\${createLink(uri: '/')}">Home</a>
    </span>
    <span class="menuButton">
      <g:link class="create" action="create">
        <g:message code="default.new.label" args="[entityName]" />
      </g:link>
    </span>
  </div>
</g:if>

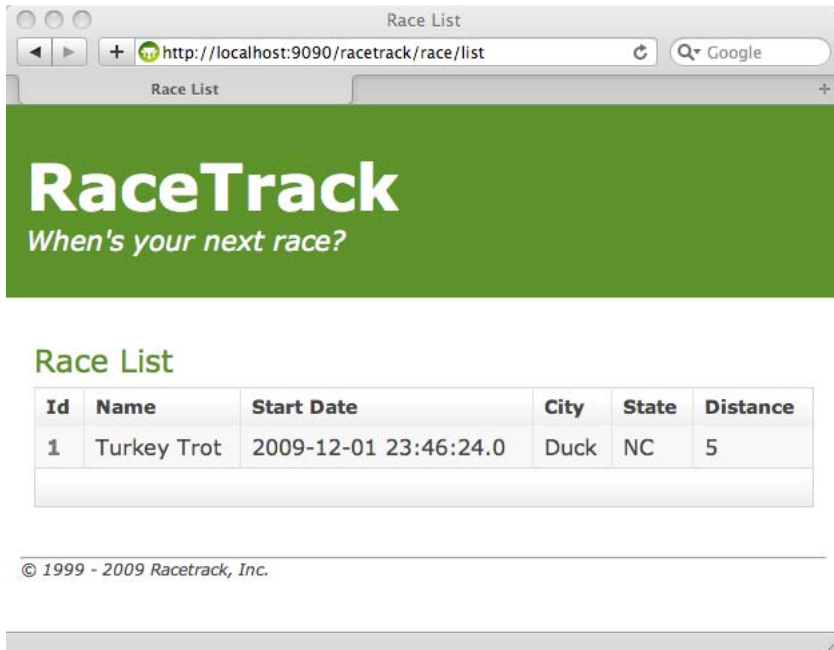
```

First of all, don't worry that we haven't created any Users yet, let alone created the login infrastructure that would put them in session scope after they successfully log in. And don't worry that your User doesn't have an `isAdmin()` method. (Does this sound like foreshadowing? It should. We'll put all of that in place in the next chapter.) The important thing to recognize for now is that you have successfully hidden the navigation bar for all non-administrative users. Make sense?

*Quick Note:* Here's one more bit of implementation trivia. When you are dealing with these GSP templates, sometimes you want the code in the `${ }` to execute when the template is generated (e.g., when you type `grails generate-all` or `grails generate-views`). Other times, you want the code in the `${ }` to execute when the view is requested in the running application. The difference is **generation-time** execution versus **run-time** execution.

If you look closely at the `list.gsp` template, you can see both in action. In `<g:if test="\${session?.user?.admin}">`, the backslash in front of the dollar sign tells Grails to execute this code at run-time. Other places, you'll see code like `var="\${propertyName}"`. In that instance, `propertyName` will be resolved when the template is generated.

Restart Grails once again, and visit any of the scaffolded `list` views for `Race`, `Runner`, or `Registration`. The navigation bar should be hidden from view.



Of course, if you visit <http://localhost:9090/racetrack/user/list>, you'll still see the navigation bar. You aren't scaffolding the views for `User` – you typed `grails generate-all User` just a moment ago. If you generate the views again, overwriting the current views, your new template will be used. The moral to this story is to install the templates early in the development process and customize them before you generate a bunch of code.

You might also want to put this change in place for `create.gsp`, `show.gsp`, and `edit.gsp` now that you're convinced that it works for `list.gsp`.

We covered a lot of ground in this chapter. You learned that GSPs are nothing more than simple HTML files with a few Grails tags (like `<g:if>`) scattered around. You learned that SiteMesh allows you to define common content for inclusion on every page (e.g., the Grails logo) by meshing `grails-app/views/layouts/main.gsp` with the individual GSP templates. For more fine-grained reuse, you can build

partial templates by creating files with a leading underscore in the name and using the `<g:render>` tag to embed them in another template. For the finest-grained reuse, you learned how to create a custom TagLib. And finally, you installed the Grails templates so that you could customize nearly every aspect of generated and scaffolded code in Grails.



## Free Online Version

Support this work, buy the print copy:

<http://www.infoq.com/minibooks/grails-getting-started>

# 9

## Security

In this chapter, we look at the basics of authentication (who are you?) and authorization (what do you have permission to do?). We'll put together a custom codec to encrypt the passwords as they are stored in the database. We'll explore both interceptors and filters, and much more.

### Implementing User Authentication

Take a look at the `User` domain class you created earlier. It has everything we need to put some basic security in place. The `login` and `password` fields will be used to *authenticate* the user – force them to identify themselves when they try to do something like change their personal information. The `role` field will be used later for *authorization* – deciding if they are allowed to do the action they are attempting.

```
class User {
    String login
    String password
    String role = "user"

    static constraints = {
        login(blank:false, nullable:false, unique:true)
        password(blank:false, password:true)
        role(inList:["admin", "user"])
    }

    String toString(){
        login
    }
}
```

Remember the check for the `admin` role you added to the `list.gsp` template in the *Groovy Server Pages* chapter? While you're here, why don't you add a convenience method called `isAdmin()`. As you saw, it's much cleaner to call `user.isAdmin` than `user.role == admin`.

```

class User {
    String login
    String password
    String role = "user"

    static constraints = {
        login(blank:false, nullable:false, unique:true)
        password(blank:false, password:true)
        role(inList:["admin", "user"])
    }

    static transients = ['admin']

    boolean isAdmin(){
        return role == "admin"
    }

    String toString(){
        login
    }
}

```

There's just one hitch when you add an accessor or mutator method (`get`, `set`, or in this case, `is`) that doesn't correspond to an actual field. It tricks GORM into looking for that field in the database table. (Hey, nobody's perfect!)

To set GORM straight, create a static `transients` list and add `admin` to it. The `transients` list is a list of fields that specifically should not be persisted back to the database. Even though there technically isn't an `admin` field, adding it to the list gets GORM off of our back.

With the admin check in place, the next thing we should do is add a couple of actions the `UserController` that will allow users to login and logout.

```

class UserController {

    def login = {}
    def logout = {}
    def authenticate = {}

    // ...
}

```

Let's tackle logging in first. Knowing what you know about the Grails lifecycle, visiting <http://localhost:9090/racetrack/user/login> will ultimately display `grails-app/views/user/login.gsp`, after passing through the login action in `UserController` first. (The `logout` and `authenticate` closures will get filled in later in the chapter, but it's

perfectly OK to leave the login closure empty. If there's nothing to do, it'll simply render the GSP as expected.)

Rather than creating the login GSP from scratch, let's borrow the existing create.gsp file and make some adjustments. Copy create.gsp to login.gsp and modify it like this:

```
<html>
  <head>
    <meta http-equiv="Content-Type"
      content="text/html; charset=UTF-8"/>
    <meta name="layout" content="main" />
    <title>Login</title>
  </head>
  <body>
    <div class="body">
      <h1>Login</h1>
      <g:if test="${flash.message}">
        <div class="message">${flash.message}</div>
      </g:if>

      <g:form action="authenticate" method="post" >
        <div class="dialog">
          <table>
            <tbody>
              <tr class="prop">
                <td valign="top" class="name">
                  <label for="login">Login:</label>
                </td>
                <td valign="top">
                  <input type="text"
                    id="login" name="login"/>
                </td>
              </tr>

              <tr class="prop">
                <td valign="top" class="name">
                  <label for="password">Password:</label>
                </td>
                <td valign="top">
                  <input type="password"
                    id="password" name="password"/>
                </td>
              </tr>
            </tbody>
          </table>
        </div>
        <div class="buttons">
          <span class="button">
            <input type="submit" value="Login" />
          </span>
        </div>
      </g:form>
    </div>
  </body>
</html>
```

Don't worry – it's not nearly as complicated as it might look at first glance. To start, change the `<title>` and `<h1>` values from Create User to Login. Next, change the form action to `authenticate`. (This is the action in `UserController` that you'll implement in just a moment.) You can simplify the `<td>` and `<input>` elements for login and password by removing all of the code that handles validation errors – for a simple login form, they'll never get used. Notice that you can remove the `role` field altogether. And finally, change the label on the submit button to Login.

When the user fills in the form and clicks the Login button, the form will be submitted to the `authenticate` action. Add this code to `UserController`.

```
def authenticate = {
  def user =
    User.findByLoginAndPassword(params.login,
                                params.password)

  if(user){
    session.user = user
    flash.message = "Hello ${user.login}!"
    redirect(controller:"race", action:"list")
  }else{
    flash.message =
      "Sorry, ${params.login}. Please try again."
    redirect(action:"login")
  }
}
```

I'll give you three guesses as to what `User.findByLoginAndPassword()` does. If you guessed, "Uh, perhaps it will find the user by login and password," give yourself a pat on the back. This is more GORM metaprogramming magic. You can `findBy` a single field, two fields using `and`, or two fields using `or`. The field names are specific to the class you are searching on, so `Race.findByNameAndCity()` is a valid query as well.

If you want to return a list of results, try `findAllBy` instead of `findBy`, as in `Race.findAllByState()`.

(For more on the dynamic finder methods that GORM offers, see <http://grails.org/DomainClass+Dynamic+Methods>.)

If the `User` record is found, it is added to the `session`, a friendly message is added to `flash` scope, and the user is redirected to the `list` action in the `RaceController`. If not, the user is redirected back to the login action.



To try this out, we should create a few sample users in `grails-app/conf/BootStrap.groovy`.

```
import grails.util.GrailsUtil

class BootStrap {
    def init = { servletContext ->
        switch(GrailsUtil.environment){
            case "development":

                def admin = new User(login:"admin",
                                    password:"wordpass",
                                    role:"admin")

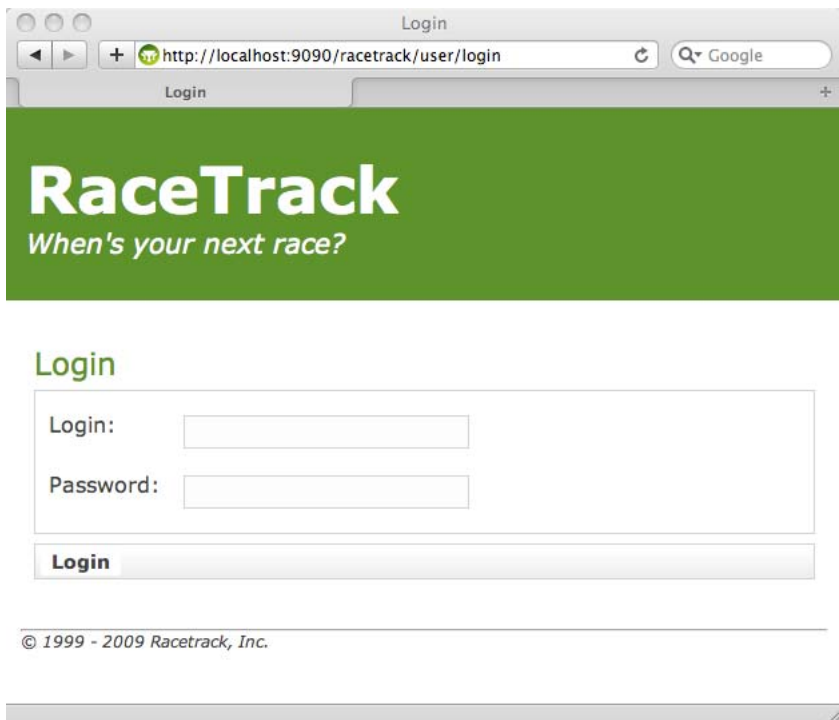
                admin.save()
                if(admin.hasErrors()){
                    println admin.errors
                }

                def jdoe = new User(login:"jdoe",
                                    password:"password",
                                    role:"user")

                jdoe.save()
                if(jdoe.hasErrors()){
                    println jdoe.errors
                }

                // ...
        }
    }
}
```

With the login form in place, the authenticate action ready and waiting, and a couple of users added for testing, let's take our new authentication system out for a spin. Start Grails and visit <http://localhost:9090/racetrack/user/login>.



The screenshot shows a web browser window with the title "Login". The address bar displays "http://localhost:9090/racetrack/user/login". The page features a green header with the text "RaceTrack" and the tagline "When's your next race?". Below the header, the word "Login" is displayed in green. The login form consists of two input fields: "Login:" and "Password:". A "Login" button is located below the password field. At the bottom of the page, a copyright notice reads "© 1999 - 2009 Racetrack, Inc.".

Login

http://localhost:9090/racetrack/user/login

Google

Login

**RaceTrack**  
*When's your next race?*

Login

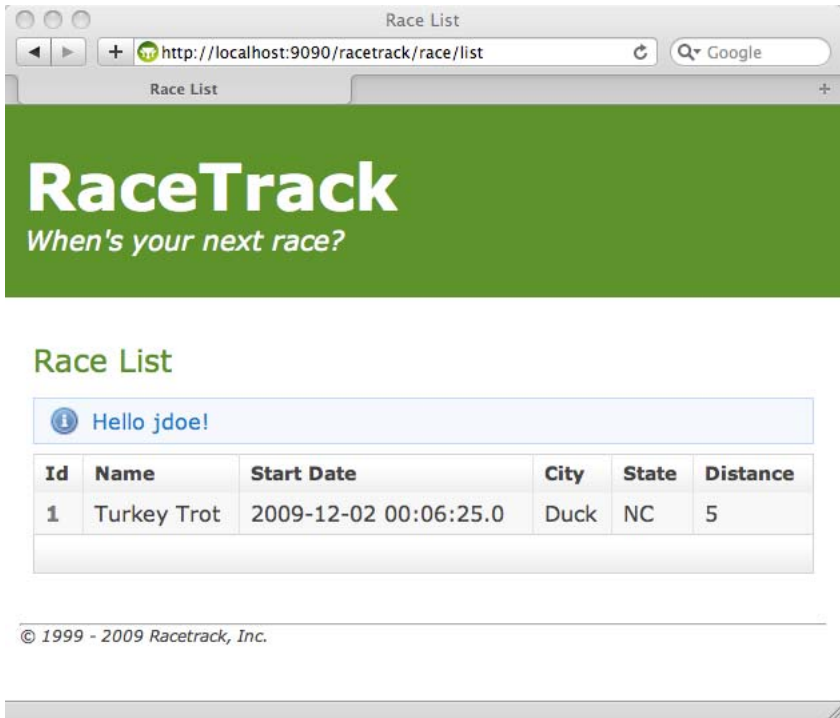
Login:

Password:

Login

© 1999 - 2009 Racetrack, Inc.

So far, so good. Try the happy path first – type `jdoe` and `password`. Do you land on the race list page?




Race List

http://localhost:9090/racetrack/race/list

RaceTrack  
*When's your next race?*

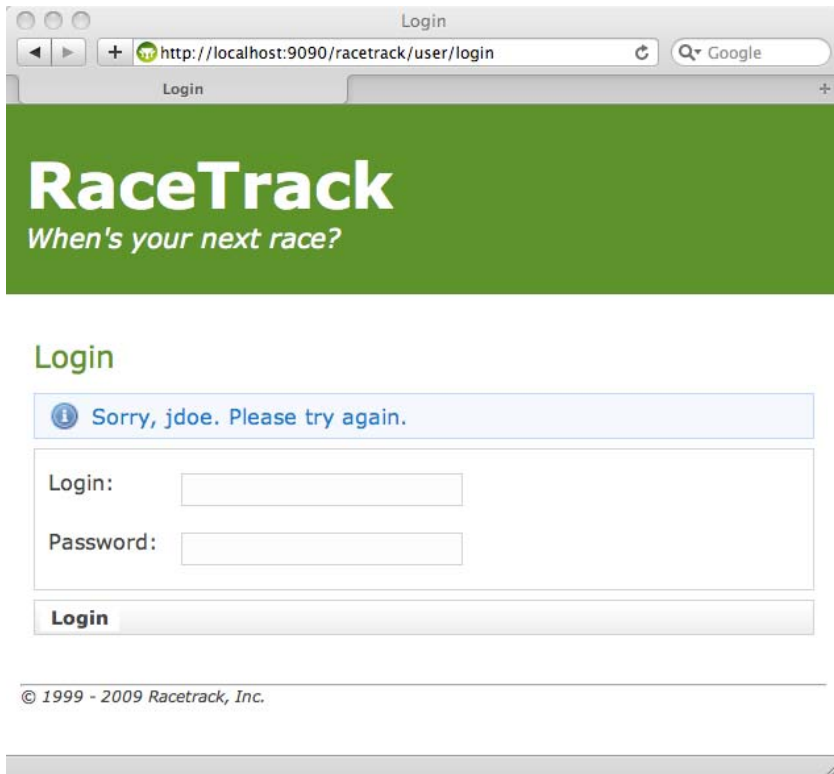
### Race List

 Hello jdoe!

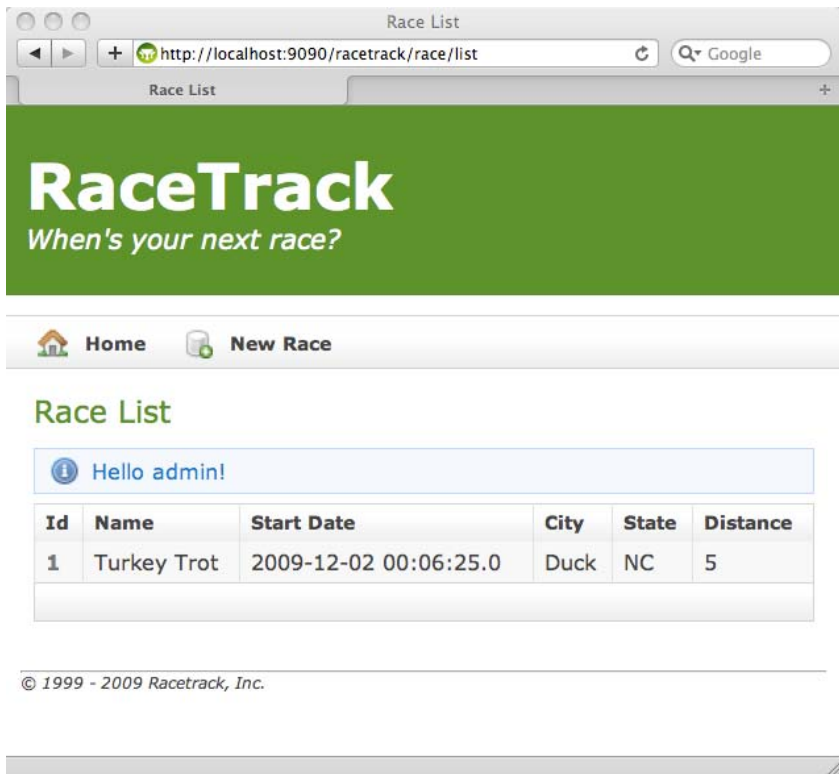
Id	Name	Start Date	City	State	Distance
1	Turkey Trot	2009-12-02 00:06:25.0	Duck	NC	5

© 1999 - 2009 Racetrack, Inc.

Great! Now trying logging in with a bad password.



All right, now for the trifecta. Log in as admin using wordpass as the password.



Did the navigation bar appear? Excellent!

To wrap up this section, add the following code to UserController to handle logging out:

```
def logout = {
  flash.message = "Goodbye ${session.user.login}"
  session.user = null
  redirect(action:"login")
}
```

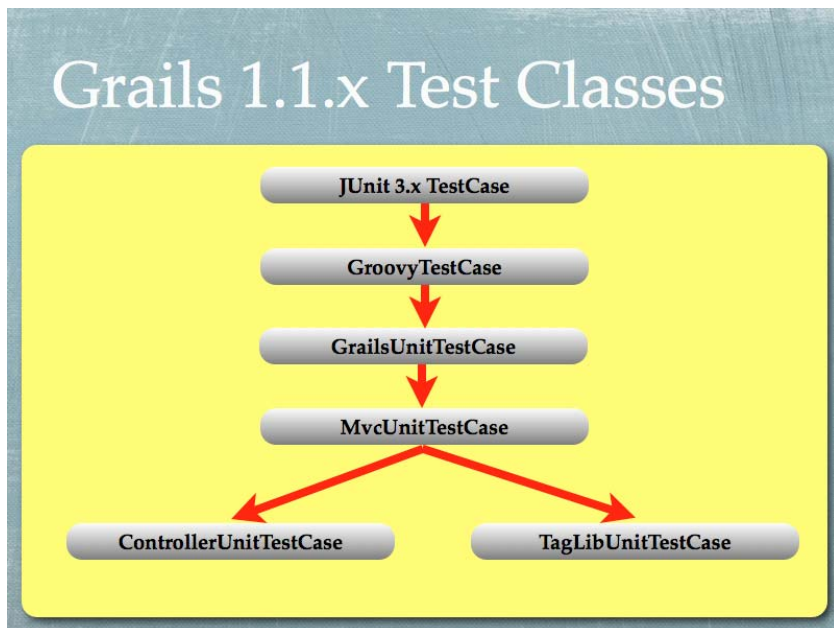
You can test this action now if you'd like, but we'll set up a TagLib in just a moment that will take full advantage of both the login and logout actions.

Now that you are convinced that everything works, wouldn't you feel better if there were a couple of unit tests in place? Of course you would...

## Unit Testing Controllers

Remember the unit and integration tests you wrote for the `Race` class? One used a `GrailsUnitTestCase`, and the other used a `GroovyTestCase`. Both extend JUnit 3.x `TestCase`. Looking at the test cases provided for the Controllers, you should see another type of `TestCase` – `ControllerUnitTestCase`.

Here's the inheritance hierarchy that was introduced in Grails 1.1:



So, what accounts for this Cambrian explosion of `TestCase` classes? The answer is simple: mocking. Integration tests are expensive to run – they require the web server to be running, all of the metaprogramming to be in place, the database to be initialized, and so on. `GrailsUnitTestCase` and its offspring offer mock services that replicate the services available in integration tests, without incurring the overhead of running the real services.

A `GrailsUnitTestCase` makes three convenience methods available: `mockForConstraintsTests()`, `mockDomain()`, and `mockLogging()`. Normally you would have to create an integration test to write assertions against this functionality. And you still can, if you'd like. But knowing that unit tests run much faster than integration tests, it's nice to be able to mock this behavior out.

For example, suppose that you'd like to test the validation on the `User` class.

```
import grails.test.*

class UserTests extends GrailsUnitTestCase {

    void testSimpleConstraints() {
        def user = new User(login:"someone",
                           password:"blah",
                           role:"SuperUser")
        // oops--role should be either 'admin' or 'user'
        // will the validation pick that up?
        assertFalse user.validate()
    }
}
```

If you run this unit test without mocking out the constraints, you'll get an interesting failure.

```
$ grails test-app
Environment set to test

Starting unit tests ...
Running tests of type 'unit'
-----
Running 8 unit tests...
Running test FooterTagLibTests...PASSED
Running test RaceControllerTests...PASSED
Running test RaceTests...PASSED
Running test RegistrationControllerTests...PASSED
Running test RegistrationTests...PASSED
Running test RunnerControllerTests...PASSED
Running test RunnerTests...PASSED
Running test UserTests...
               testSimpleConstraints...FAILED
Tests Completed in 734ms ...
-----
Tests passed: 7
Tests failed: 1
-----

Starting integration tests ...
...

[junitreport] Processing
Tests FAILED - view reports in target/test-reports.
```

Look in the `test/reports` directory for the HTML report.

Unit Test Results.

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

**Class UserTests**

Name	Tests	Errors	Failures	Time(s)	Time Stamp	Host
UserTests	1	1	0	0.149	2009-09-03T06:12:51	scott-davis2s-macbook-pro.local

**Tests**

Name	Status	Type
testSimpleConstraints	Error	No signature of method: User.validate() is applicable for argument types: () values: []  groovy.lang.MissingMethodException: No signature of method: User.validate() is applicable for argument types: () values: [] at UserTests.testSimpleConstraints(UserTests.groovy:18) at _GrailsTest_groovy\$_run_closure4.doCall(_GrailsTest_groovy:223) at _GrailsTest_groovy\$_run_closure4.call(_GrailsTest_groovy) at _GrailsTest_groovy\$_run_closure2.doCall(_GrailsTest_groovy:167)

The error is “No signature of method: User.validate().” This confirms that the `validate()` method gets metaprogrammed onto domain classes for integration tests, but not for unit tests. So you have two choices: either move this test from the unit to the integration folder, or mock out the constraints.

Keep in mind that there is absolutely nothing wrong with integration tests. If we run this test in a live, running Grails application, you don’t have to do any mocking at all – everything behaves exactly as it would in production. The only reason I’m advocating mocking out this functionality is speed, speed, and speed. The sad truth is that if your tests take longer to run, you’ll be less likely to run them. The mocking capabilities of `GrailsUnitTestCase` are there to speed up the testing process for you, removing *yet another* lame excuse that you might try to use to not test your code.



In that spirit, add a couple of lines to your test to mock out the constraints test:

```
import grails.test.*

class UserTests extends GrailsUnitTestCase {
    void testSimpleConstraints() {
        mockForConstraintsTests(User)
        def user = new User(login:"someone",
                           password:"blah",
                           role:"SuperUser")
        // oops--role should be either 'admin' or 'user'
        // will the validation pick that up?
        assertFalse user.validate()
        assertEquals "inList", user.errors["role"]
    }
}
```

The `mockForConstraintsTest()` method tells the unit test to metaprogram the methods for constraints evaluation onto the `User` class. The last line asserts that the `inList` validation failed for the `role` field.

To run all tests – both unit and integration – type `grails test-app`. If you'd like to run just unit tests, type `grails test-app -unit`. For just the integration tests, type `grails test-app -integration`. To run just the unit test for `User`, type `grails test-app User -unit`.

```
$ grails test-app User -unit

Starting unit tests ...
Running tests of type 'unit'
-----
Running 1 unit test...
Running test UserTests...PASSED
Tests Completed in 466ms ...
-----
Tests passed: 1
Tests failed: 0
```

So now that you understand `mockForConstraintsTests()`, let's have a look at `mockDomain()`. It takes mocking one step further – it allows you to mock out the entire database layer. You'll need a database (mocked or real) in order to test the unique constraint on the `login` field. We'll use a mocked database for our purposes here.

Add this test to `UserTests`:

```
void testUniqueConstraint(){
    def jdoe = new User(login:"jdoe")
    def admin = new User(login:"admin")
    mockDomain(User, [jdoe, admin])

    def badUser = new User(login:"jdoe")
    badUser.save()
    assertEquals 2, User.count()
    assertEquals "unique", badUser.errors["login"]

    def goodUser = new User(login:"good",
                             password:"password",
                             role:"user")

    goodUser.save()
    assertEquals 3, User.count()
    assertNotNull User.findByLoginAndPassword(
        "good", "password")
}
```

The `mockDomain()` method creates a mock table with two users in it. Trying to save a user with a login that conflicts with an existing user fails due to the unique constraint. The `goodUser`, however, is saved without error. (Did you notice that `mockDomain()` gives you all of the GORM dynamic finder goodness as well?)

The final `GrailsUnitTestCase` method – `mockLogging()` – injects a mock log object into your class and writes the output to `System.out` instead of relying on `Log4J` to be configured correctly.

But didn't we start out wanting to test our new `authenticate` action in `UserController`? Here's where the `ControllerUnitTestCase` comes in handy. If it doesn't already exist, create `UserControllerTests.groovy` in the `test/unit` directory.

```
import grails.test.*

class UserControllerTests extends ControllerUnitTestCase
{
    void testSomething() {

    }
}
```

A `ControllerUnitTestCase` can do everything a `GrailsUnitTestCase` can do, and then some.

For example, you can verify that redirects happen as expected. You know that the `index` action in `UserController` should redirect to the `list` action:

```
class UserController {
  def index = {
    redirect action:"list", params:params
  }
}
```

Here is the test to prove it:

```
import grails.test.*

class UserControllerTests extends ControllerUnitTestCase
{
  void testIndex() {
    controller.index()
    assertEquals "list",
                  controller.redirectArgs["action"]
  }
}
```

A `UserController` instance is passed to each test method in a variable named `controller`. In this simple test, you invoke the `index()` action and assert that the `list` action is the value of the `action` key in the redirect `HashMap`.

Type `grails test-app UserController -unit` to run this test.

Here's a slightly more involved example. Remember that in a URL like <http://localhost:9090/racetrack/user/show/2>, the number 2 shows up in the query string as `params.id`.

```
class UserController{
  def show = {
    def userInstance = User.get(params.id)
    if (!userInstance) {
      flash.message="{message(code:'default.not.found.message',
                             args: [message(code: 'user.label',
                             default: 'User'), params.id])}"
      redirect(action: "list")
    }
    else {
      [userInstance: userInstance]
    }
  }
}
```

To test this action, you need to ensure that the params HashMap is populated correctly before you call the action.

```
class UserControllerTests extends ControllerUnitTestCase
{
    void testShow(){
        def jdoe = new User(login:"jdoe")
        def suziq = new User(login:"suziq")
        mockDomain(User, [jdoe, suziq])

        controller.params.id = 2
        def map = controller.show()
        assertEquals "suziq", map.userInstance.login
    }
}
```

Knowing how to stuff the right values into params, you can now write a test for the controller's authenticate action:

```
void testAuthenticate(){
    def jdoe = new User(login:"jdoe",
                        password:"password")
    mockDomain(User, [jdoe])

    controller.params.login = "jdoe"
    controller.params.password = "password"
    controller.authenticate()
    assertNotNull controller.session.user
    assertEquals "jdoe", controller.session.user.login

    controller.params.password = "foo"
    controller.authenticate()
    assertTrue controller.flash.message.startsWith(
        "Sorry, jdoe")
}
```

You can work with the session and flash HashMaps coming out in the same way that you work with the params HashMap going in.

Here is the full list of mocked elements for a ControllerUnitTestCase:

- controller.request
- controller.response
- controller.session
- controller.flash
- controller.redirectArgs
- controller.renderArgs
- controller.template
- controller.modelAndView

OK, now that you know how to test your `authenticate` action, let's get back to securing this application. Next up: getting rid of the plain-text passwords stored in the database.

## Creating a Password Codec

A codec (short for coder-decoder) is a way to transform a `String`. Grails offers a number of convenient codecs already metaprogrammed onto all `String` instances.

For example:

- `"<p>Hello</p>".encodeAsHTML()` returns `<p>Hello</p>`
- `"You & Me".encodeAsURL()` returns `You+%26+Me`
- `"ABC123".encodeAsBase64()` returns `QUJDMTIz`

There are `decodeHTML()`, `decodeURL()` and `decodeBase64()` methods that do the reverse transformation as well.

If you want to create your own codec, simply create a file that ends with `Codec` in the `grails-app/utils` directory. For example, suppose that you want to convert spaces to underscores. To do so, create `UnderscoreCodec.groovy` and add the following code:

```
class UnderscoreCodec {
    static encode = {target->
        target.replaceAll(" ", "_")
    }

    static decode = {target->
        target.replaceAll("_", " ")
    }
}
```

You can then call `"Hello World".encodeAsUnderscore()` and `"Hello_World".decodeUnderscore()` throughout your application.

So how does that help us with encrypting our passwords? Create `grails-app/utils/SHACodec.groovy` and add the following code:

```
import java.security.MessageDigest

class SHACodec{
    static encode = {target->
        MessageDigest md = MessageDigest.getInstance('SHA')
        md.update(target.getBytes('UTF-8'))
        return new String(md.digest()).encodeAsBase64()
    }
}
```

As you'll see in just a moment, all we really need to do is encode the password. When the plain-text password comes in, we simply hash it and compare the results with the hashed value stored in the database. The lack of a decode closure means that there is no danger of the encoded password in the database being decoded into its plain-text form – either by accident or on purpose.

There are now two places to apply the newly created codec. First, add a `beforeInsert` closure to `User.groovy`:

```
class User {
    String login
    String password
    String role = "user"

    static constraints = {
        login(blank:false, nullable:false, unique:true)
        password(blank:false, password:true)
        role(inList:["admin", "user"])
    }

    static transients = ['admin']

    boolean isAdmin(){
        return role == "admin"
    }

    def beforeInsert = {
        password = password.encodeAsSHA()
    }

    String toString(){
        login
    }
}
```

This ensures that the password is always encrypted, regardless of whether you go through `Bootstrap.groovy`, the `create` action in `UserController`, or anything else. Start up Grails and then verify that the passwords are hashed in the database.

```
mysql> select * from user;
```

id	login	password	role
1	admin	KTkJTzWjut8qiQdougNPpesW6V4=	admin
2	jdoe	W6ph5Mm5Pz8GgiULbPgZG37mj9g=	user

Next, tweak the `authenticate` action to check for the hashed password instead of the plain-text one.

```
class UserController {

    def authenticate = {
        def user = User.findByLoginAndPassword(
            params.login, params.password.encodeAsSHA())
        if(user){
            session.user = user
            flash.message = "Hello ${user.login}!"
            redirect(controller:"race", action:"list")
        }else{
            flash.message =
                "Sorry, ${params.login}. Please try again."
            redirect(action:"login")
        }
    }
}
```

No more plain-text passwords in our application, eh? Feel free to test this out in your running Grails application. I'll wait right here.

Of course, you'll want to tweak your `test/unit/UserControllerTests.groovy` file now as well. After all, you wouldn't want to leave failing tests laying around to mess up your otherwise-perfect application, would you? Remember that this is a unit test, and Codecs don't get metaprogrammed on to Strings in unit tests.

You need to do three things to upgrade your unit test: import the package where the Grails Codecs live, manually metaprogram the codecs on to String in the `setUp()` method, and adjust your mocked password to be SHA-encoded in `testAuthenticate()`.

```

import grails.test.*
import org.codehaus.groovy.grails.plugins.codecs.*

class UserControllerTests extends ControllerUnitTestCase
{
    protected void setUp() {
        super.setUp()
        String.metaClass.encodeAsBase64 = {->
            Base64Codec.encode(delegate)
        }
        String.metaClass.encodeAsSHA = {->
            SHACodec.encode(delegate)
        }
    }

    // ...
    void testAuthenticate(){
        def jdoe = new User(login:"jdoe",
            password:"password".encodeAsSHA())
        mockDomain(User, [jdoe])

        controller.params.login = "jdoe"
        controller.params.password = "password"
        controller.authenticate()
        assertNotNull controller.session.user
        assertEquals "jdoe", controller.session.user.login

        controller.params.password = "foo"
        controller.authenticate()
        assertTrue
        controller.flash.message.startsWith("Sorry, jdoe")
    }
}

```

Type `grails test-app` to make sure that all tests are passing now.

Next, let's make it easier for users to log in and log out by providing a link in the header.



## Creating an Authorization TagLib

Most public websites have an unobtrusive login link in the upper right corner of the header. We can do the same using the TagLib skills we picked up in the *Groovy Server Pages* chapter.

Type `grails create-tag-lib Login`, and add the following code to `grails-app/taglib/LoginTagLib.groovy`:

```
class LoginTagLib {
    def loginControl = {
        if(request.getSession(false) && session.user){
            out << "Hello ${session.user.login} "
            out << """[${link(action:"logout",
                controller:"user") {"Logout"}}]"" "
        } else {
            out << """[${link(action:"login",
                controller:"user") {"Login"}}]"" "
        }
    }
}
```

The next step is to add it to `grails-app/views/layout/_header.gsp`:

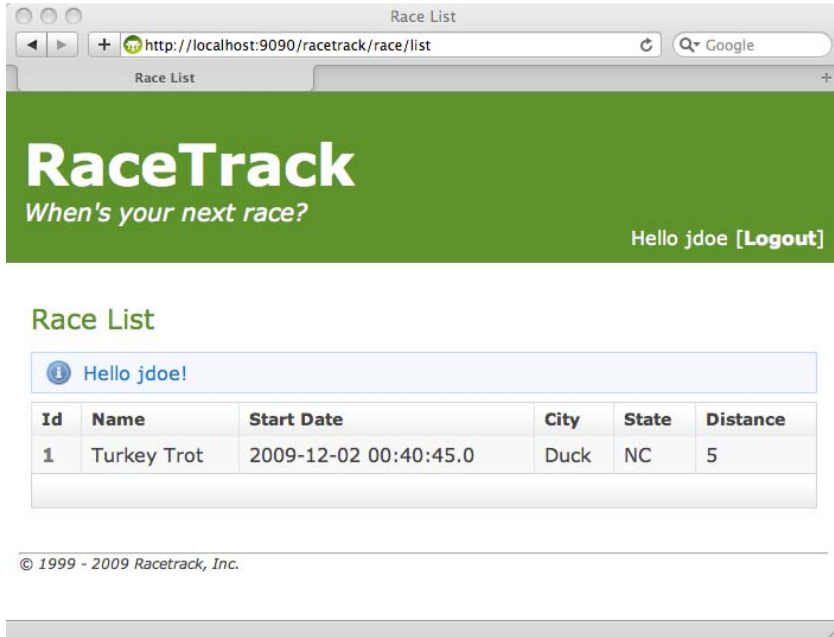
```
<div id="header">
    <p>
        <a class="header-main" href="${resource(dir:')}">
            RaceTrack
        </a>
    </p>
    <p class="header-sub">When's your next race?</p>
    <div id="loginHeader">
        <g:loginControl />
    </div>
</div>
```

Add a little CSS to `web-app/css/main.css` to make it look pretty:

```
#loginHeader {
    float: right;
    color: #fff;
}

#loginHeader a{
    color: #fff;
}
```

And restart Grails once again to see how it looks:



That was easy, wasn't it? You could take this a step further, adding your own `<g:loggedIn>` tag, `<g:isAdmin>` tag, and so on. But I think that you get the gist of things.

Next, let's lock down the `UserController` so that only admins can create new users.

## Leveraging the beforeInterceptor

We could litter each of the actions in `UserController` with repetitive / boilerplate `if(session.user.admin)` checks, but there's an easier way to ensure that this functionality is only available to admin users. We can set up a `beforeInterceptor` block that – as the name implies – gets called before each action is invoked. (Grails offers an `afterInterceptor` as well.)

Add the following to `UserController`:

```
class UserController {

    def beforeInterceptor = [action:this.&debug]

    def debug(){
        println "DEBUG: ${actionUri} called."
        println "DEBUG: ${params}"
    }

    def logout = {
        flash.message = "Goodbye ${session.user.login}"
        session.user = null
        redirect(action:"login")
    }

    // ...
}
```

Notice the subtle but important difference between `debug` and `logout`. Closures like `logout` are exposed out to the end user in the URL. Methods like `debug()` are private and can only be called by other closures or methods in the class. (You can tell that `debug()` is a method by the trailing parentheses.)

The ampersand in the `beforeInterceptor` is a method pointer, supplied by Groovy. This means, in essence, that the `debug()` method will be called before each action is invoked.

Start Grails and visit <http://localhost:9090/racetrack/user>. You should see the following output in the console window where you started Grails:

```
DEBUG: /user/index called.
DEBUG: [controller:user]
DEBUG: /user/list called.
DEBUG: [action:list, controller:user]
```

As expected, the implicit call to `index` is redirected to `list`, resulting in two calls to the `debug()` method.

Now, let's point the `beforeInterceptor` to a slightly more useful method. Add the following code to `UserController`:

```
class UserController {

  def beforeInterceptor = [action:this.&auth,
                          except:['login', 'logout', 'authenticate']]

  def auth() {
    if(!session.user) {
      redirect(controller:"user", action:"login")
      return false
    }

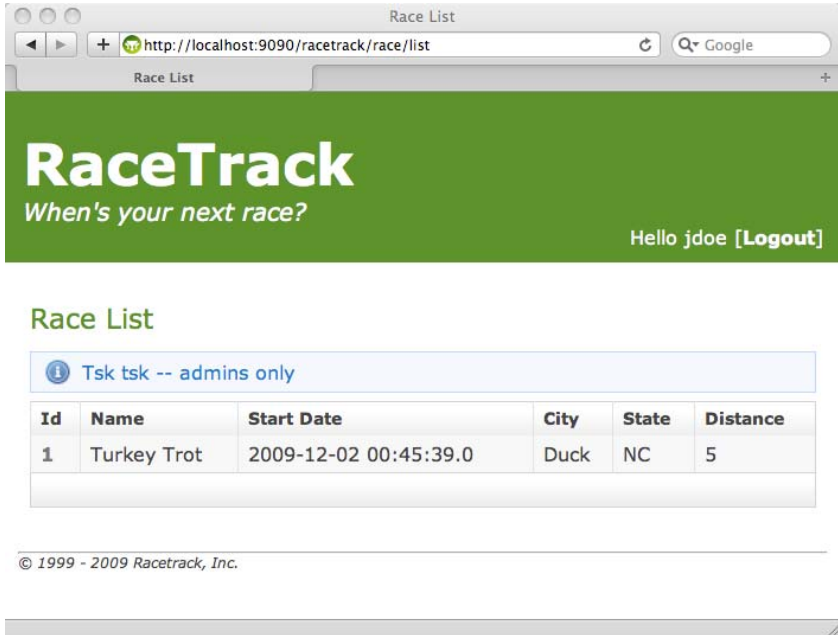
    if(!session.user.admin){
      flash.message = "Tsk tsk-admins only"
      redirect(controller:"race", action:"list")
      return false
    }
  }

  // ...
}
```

Notice this time that we point the `beforeInterceptor` to the `auth()` method, where it first checks to see if you are logged in, and then checks to see if you are an admin (since we only want admins to be able to manage the user data).

The `except` list allows unfettered access to the `login`, `logout`, and `authenticate` actions (i.e., this filter applies to all actions in this controller *except* these three actions). (Instead of `except`, you can also use `only` to declare an explicit list of actions to which the filter will apply.)

Start Grails and visit <http://localhost:9090/racetrack/user>. Verify that the on the first visit, you are redirected to the login screen. Next, login using `jdoe/password`. Since that user doesn't have admin privileges, you should be redirected and gently scolded:



Finally, login using admin/wordpress and verify that that, as an admin, you can hit any of the actions in `UserController` that you'd like.

The last thing we'll do in this chapter is step up from working with individual controllers to applying security across all controllers. To do this, we need to create a Filter.

## Leveraging Filters

The `beforeInterceptor` allows you to intercept requests on a per-Controller basis. Filters allow you to do the same thing on an application-wide basis. Type `grails create-filters Admin` at the command prompt. Then, open `grails-app/conf/AdminFilters.groovy` in a text editor.

```
class AdminFilters {
    def filters = {
        all(controller: '*', action: '*') {
            before = {
            }
            after = {
            }
            afterView = {
            }
        }
    }
}
```

Inside of the `filters` closure, a default filter named `all` is defined. As you can see, it will intercept calls to any controller, any action. Inside `all`, you can populate `before` (invoked before the action is executed, exactly like `beforeInterceptor`), `after` (invoked after the action but before the view is rendered), or `afterView` (invoked after the view is rendered).

You can have as many entries as you'd like inside the `filters` closure. In this case, I renamed the `all` filter to `adminOnly` to better document its purpose – any call that can modify a record is limited to a logged-in admin. I also removed the unused `after` and `afterView` closures. Here is the result:

```
class AdminFilters {
    def filters = {
        adminOnly(controller: '*',
            action: "(create|edit|update|delete|save)") {
            before = {
                if(!session?.user?.admin){
                    flash.message = "Sorry, admin only"
                    redirect(controller:"race", action:"list")
                    return false
                }
            }
        }
    }
}
```

A common gotcha is the way you define multiple actions for a filter. In a `beforeInterceptor`, you pass in a list of actions. In a filter, they specify the actions in a *regular expression*. Other than that difference, you can do all of the same things in a filter than you could do in an interceptor.

## Security Plugins

We haven't discussed plugins yet. (Just wait – we'll tackle them in-depth in the next chapter), but Grails makes adding whole new swaths of functionality as easy as typing `grails install-plugin`. (You can type `grails list-plugins` or visit <http://grails.org/plugins> to get an idea of the various types of plugins available.)

My goal in this chapter was to show you the core mechanisms you can use to secure your application – interceptors, filters, and so on. For a simple application like RaceTrack, our home-grown approach is entirely adequate. If your security needs are more sophisticated, or you need to interact with external systems – like LDAP (<http://grails.org/plugin/ldap>) or OpenID (<http://grails.org/plugin/openid>) – you might decide that installing one of the many pre-existing security plugins is a better approach.

There are many popular security plugins available. I encourage you to visit <http://grails.org/plugins> and find one that best suits your needs. Different types of websites have different requirements, so security is far from a “one size fits all” solution.

For example, if you run a public-facing website and would like to enable “self registration” with an email address, you might find the Authentication plugin (<http://grails.org/plugin/authentication>) interesting. It offers out-of-the-box support for the ubiquitous “Can you email my forgotten password to me” requests, and much more.

For a more traditional, intranet-based, closed security application, the JSecurity plugin (<http://grails.org/plugin/jsecurity>) offers the right combination of simplicity and power. It is written and supported by the core team of Grails developers, so you know that it won't be neglected in the long run.

If you already have experience with the Spring Security framework (formerly known as ACEGI), there is a plugin for you (<http://grails.org/plugin/acegi>) as well.

Regardless of which plugin you choose, rest assured that they all use the same techniques you learned in this chapter. They provide a User account for authentication, some form of roles for authorization, and TagLibs, Interceptors, and Filters for herding the users in the right direction and keeping them out of the places they shouldn't be.



### Free Online Version

Support this work, buy the print copy:

<http://www.infoq.com/minibooks/grails-getting-started>

# 10

## Plugins, Services, and Deployment

In this chapter, we'll explore the Grails plugin ecosystem. Along the way, you'll learn about one more core Grails technology: Services. And finally, you'll create a WAR file that is ready to be deployed to Tomcat, JBoss, GlassFish, or any other JEE application server.

### Understanding Plugins

Java developers are accustomed to mixing in new functionality by dropping JARs onto the CLASSPATH. Earlier in the book, you didn't have to write your own MySQL JDBC driver to talk to the database – you simply copied the appropriate JAR file into the `lib` directory.

Grails plugins are like JARs on steroids. They can include not only one or more JARs, but also GSPs, Controllers, TagLibs, Codecs, and more. In essence, they are mini Grails applications expressly meant to be merged with other Grails applications.

At the end of the last chapter, we touched on a number of different security plugins available at <http://grails.org/Plugins>. But plugins can be used for more than just security. There are plugins that provide alternate view layers like Flex (<http://grails.org/plugin/flex>) and the Google Web Toolkit (<http://grails.org/plugin/gwt>). There are plugins that incorporate additional testing tools like Selenium (<http://grails.org/plugin/selenium>) and Cobertura (<http://grails.org/plugin/code-coverage>). But in each case, a plugin is comprised of familiar Grails components that you are used to creating on your own.

## Installing the Searchable Plugin

---

Let's add search capabilities to RaceTrack. Knowing what you know about the GORM dynamic finders, you could probably put together a pretty decent search feature if it was limited to a single class and one or two fields.

For example, say that you wanted to provide a search form for races in a specific city. If the search form had a text field named `query`, here is the Controller action that could provide the results:

```
def search = {
    def results = Race.findAllByCity(params.query)
    return [searchResults:results]
}
```

If you wanted to search by `city` and `state`, you could adjust the query to

```
Race.findAllByCityAndState(params.city,
params.state).
```

But what if you want to search on more than two fields? (Dynamic finders are limited to one or two fields.) And what if you wanted to search for cities across multiple classes, like `Races`, `Runners`, `Sponsors`, and so on?

You could roll up your sleeves and try to write comprehensive search facilities yourself, or you could type `grails install-plugin searchable` and take advantage of someone else's hard work. Since laziness is a virtue among software developers – we call it *code reuse* to make it sound better to non-programmers – let's take the latter approach.

There's a lot of console activity when you install a new plugin. Here's a guided tour of what is going on.

```
$ grails install-plugin searchable
...

Reading remote plugin list ...
Plugin list out-of-date, retrieving..
[get] Getting: http://svn.codehaus.org/grails/trunk/grails-
plugins/.plugin-meta/plugins-list.xml
[get] To: /Users/sdavis/.grails/1.2.0/plugins-list-core.xml

Reading remote plugin list ...
[get] Getting: http://plugins.grails.org/.plugin-meta/plugins-
list.xml
[get] To: /Users/sdavis/.grails/1.2.0/plugins-list-default.xml
```

To begin, you can see that Grails places a couple of calls back to the mothership to get a current list of plugins. The core list contains required plugins – plugins that Grails cannot run without. (At the time of this writing, there is only one – the Hibernate plugin for GORM support.) The other list – default – contains all of the optional plugins like searchable.

Here is the snippet from `plugins-list-default.xml` that describes the searchable plugin.

```
<plugin latest-release="0.5.5" name="searchable">
  <release tag="RELEASE_0_5_5"
    type="svn" version="0.5.5">
    <title>Adds rich search functionality to
      Grails domain models.
      This version is recommended for JDK 1.5+</title>
    <author>Maurice Nicholson</author>
    <authorEmail>maurice@freeshell.org</authorEmail>
    <description>Adds rich search functionality to Grails
      domain models.
      Built on Compass (http://www.compass-project.org/)
      and Lucene (http://lucene.apache.org/)
      This version is recommended for JDK 1.5+
    </description>
    <documentation>
      http://grails.org/Searchable+Plugin
    </documentation>
    <file>http://plugins.grails.org/grails-searchable/
      tags/RELEASE\_0\_5\_5/grails-searchable-0.5.5.zip
    </file>
  </release>

  <!-- snip -->
</plugin>
```

As you can see, there is lots of nice metadata about the searchable plugin – who wrote it, the current version, and (most importantly) where to download it. Knowing that last bit of information, it's no surprise what Grails does next:

```
$ grails install-plugin searchable
... download core and default plugin lists

... continued ...
[get] Getting: http://plugins.grails.org/
grails-searchable/tags/RELEASE\_0\_5\_5/
grails-searchable-0.5.5.zip
[get] To: /Users/sdavis/.grails/1.2.0/plugins/
grails-searchable-0.5.5.zip
```

The plugin is downloaded to the `.grails` directory in your home directory. The next time you try to install the searchable plugin in

another project, Grails will check for the latest version and install it from the `.grails` cache instead of the web if it can.

Once the plugin is downloaded, the next step is to copy it to your project:

```
$ grails install-plugin searchable

... continued ...
[copy] Copying 1 file to
/Users/sdavis/.grails/1.2.0/projects/racetrack/plugins
Installing plug-in searchable-0.5.5
[mkdir] Created dir:
/Users/sdavis/.grails/1.2.0/projects/racetrack/plugins/search
able-0.5.5
[unzip] Expanding:
/Users/sdavis/.grails/1.2.0/plugins/grails-searchable-
0.5.5.zip into
/Users/sdavis/.grails/1.2.0/projects/racetrack/plugins/search
able-0.5.5
```

Notice that Grails doesn't copy it to your local source directory. It copies it to the `.grails` directory where your compiled code lives. We'll explore that directory in greater detail in just a moment. For now, let's finish exploring the console output.

```
$ grails install-plugin searchable

... continued ...
Thanks for installing the Grails Searchable Plugin!

Documentation is available at
http://grails.org/Searchable+Plugin

Help is available from user@grails.codehaus.org

Issues and improvements should be raised at
http://jira.codehaus.org/browse/GRAILSPUGINS

If you are upgrading from a previous release, please see
http://grails.org/Searchable+Plugin+-+Releases

Plugin searchable-0.5.5 installed
Plug-in provides the following new scripts:
-----
grails install-searchable-config
```

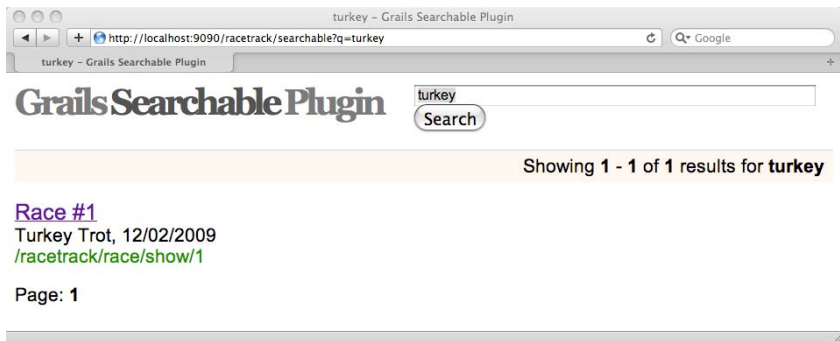
The installation concludes with a list of helpful resources for working with the plugin: a link for further documentation, a link for submitting feature requests, and an email address for requesting assistance.

You'll be happy to know that the one line installation command is matched in simplicity by the one line required to make domain classes searchable.

Open `grails-app/domain/Race.groovy` and add a single line to it:

```
class Race {
    static searchable = true
    // ...
}
```

Now fire up Grails and visit <http://localhost:9090/racetrack/searchable>. Verify that you can search for any word stored in any field in any `Race` entry. (Feel free to add a few more `Races` to make searching a bit more interesting.)



Not a bad return on investment for one line of code, eh?

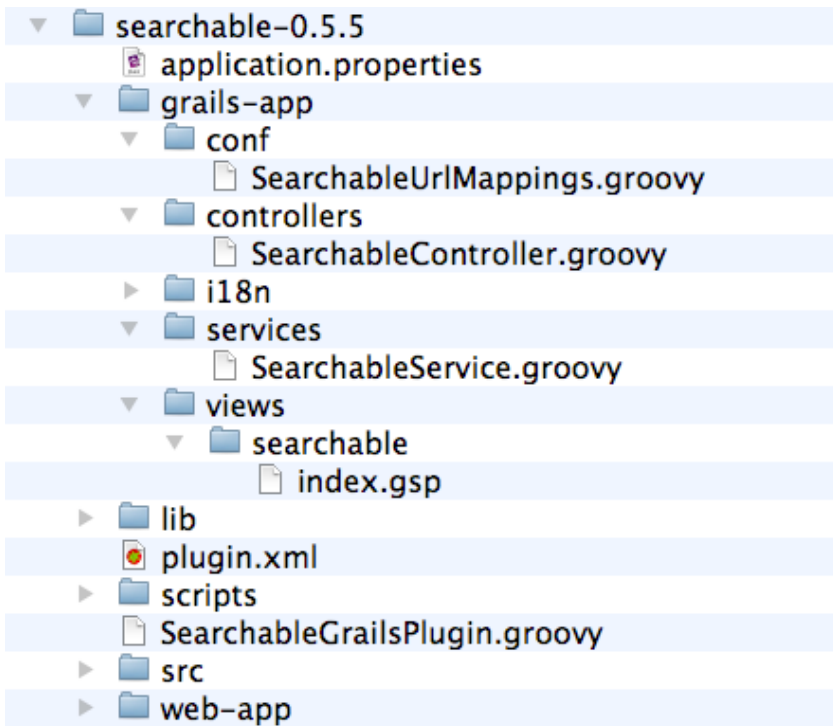
As it stands now, the search feature is open for anyone to use. That means it's probably best not to include private details from the `Runner` or `Registration` class by adding `static searchable = true` to them. If, however, you want the search feature to be more of an admin feature, you can add both private data and a bit of security to protect it. (You can add a security check for the `Searchable` controller to `grails-app/conf/AdminFilters.groovy` to easily lock this feature down.)

Now that you are convinced that the searchable plugin works as advertised, the next thing we should do is customize it a bit. The look and feel doesn't match ours, and I can see another custom `TagLib` in our immediate future.

## Exploring the Searchable Plugin

Knowing what you know about Grails, if you can call <http://localhost:9090/racetrack/searchable>, there should be a `SearchableController.groovy` file in `grails-app/controllers`. But looking in your local source tree, there is no `SearchableController` to be found.

Remember that plugins are installed in the magic `.grails` directory in your home directory. Take a look at `.grails/1.2.0/projects/racetrack/plugins`, and you should find exactly what you are looking for – a familiar Grails directory structure.



As you probably guessed, the search form can be found in `grails-app/views/searchable/index.gsp`:

```
<g:form url='[controller: "searchable", action: "index"]'
      id="searchableForm"
      name="searchableForm"
      method="get">
  <g:textField name="q" value="${params.q}" size="50"/>
  <input type="submit" value="Search" />
</g:form>
```

It submits the query to the `index` action in the `SearchableController`:

```
import
org.compass.core.engine.SearchEngineQueryParseException

class SearchableController {
  def searchableService

  def index = {
    if (!params.q?.trim()) {
      return [:]
    }
    try {
      return [searchResult:
        searchableService.search(params.q, params)]
    } catch (SearchEngineQueryParseException ex) {
      return [parseException: true]
    }
  }
  // ...
}
```

The `index` closure is clearly interested in the `params.q` field. If the parameter doesn't exist, the action returns an empty `HashMap`. If, on the other hand, the parameter exists, it returns the results of the `searchableService.search()` method call.

This begs the question, “What is a Service?” (I’m so glad you asked. Read on!)

## Understanding Services

---

Controllers are usually focused on the care and feeding of a single domain class. Services, on the other hand, are places to put business logic and behavior that is applicable to more than one domain class.

Did you notice how the `SearchableService` (stored in `grails-app/services`) got added to the `SearchableController`? The single line `def searchableService` right after the class declaration of the Controller did the trick. This is what the cool kids call ***dependency injection***, or “DI” for short. DI is what the Spring framework (another one of the core technologies of Grails) handles adroitly.

As a matter of fact, take a look at the source of `SearchableService`:

```
class SearchableService {
    boolean transactional = true
    def compass
    def compassGps
    def searchableMethodFactory

    def search(Object[] args) {
        searchableMethodFactory.
            getMethod("search").invoke(*args)
    }

    // ...
}
```

Right off the bat, you can see a couple of other classes that get injected into the service – `compass`, `compassGps`, and `searchableMethodFactory`. You can find their implementations in `src/java` and `src/groovy`.

But rather than getting bogged down in implementation details, let’s get back to the task at hand: let’s add a search box to the header of our application.

## Adding a Search Box

---

Since the only class that is searchable at this point is `Race`, we can add a `search` action to `RaceController`. (If you decide to refactor the search functionality to include more classes later on, you may want to create a more generic `SearchController` as well.)

You could inject the `SearchableService` into `RaceController` the same way it was injected into `SearchableController`, but if you are



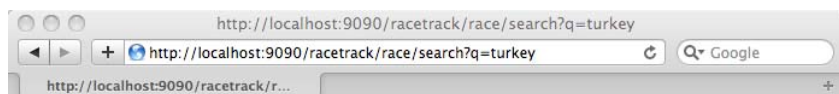
searching on a single class, there is an easier way. The searchable plugin metaprograms a `search()` method onto each domain class. That means that our first pass at the search implementation looks like this:

```
class RaceController {
  def scaffold = Race

  def search = {
    render Race.search(params.q, params)
  }
}
```

Take a moment to consider how much is happening here in so few lines of code. It's pretty amazing, isn't it?

Test it out by visiting <http://localhost:9090/racetrack/race/search?q=turkey>



```
["total":1,
"hits":org.compass.core.impl.DefaultCompassDetachedHits@7be5f7c4,
"max":10, "scores":[0.26492345], "results":[Turkey Trot, 12/02/2009],
"offset":0]
```

Well it works, but that UI is something only a developer could love. Let's wrap it up in a slightly friendlier interface.

To start, create a partial template named `_raceSearch.gsp` in `grails -app/views/layouts`.

```
<div id="search">
  <g:form url='[controller: "race", action: "search"]'
    id="raceSearchForm"
    name="raceSearchForm"
    method="get">
    <g:textField name="q" value="\${params.q}" />
    <input type="submit" value="Find a race" />
  </g:form>
</div>
```

Next, add it to the top of the `_header.gsp` template in the same directory:

```
<g:render template="/layouts/raceSearch" />

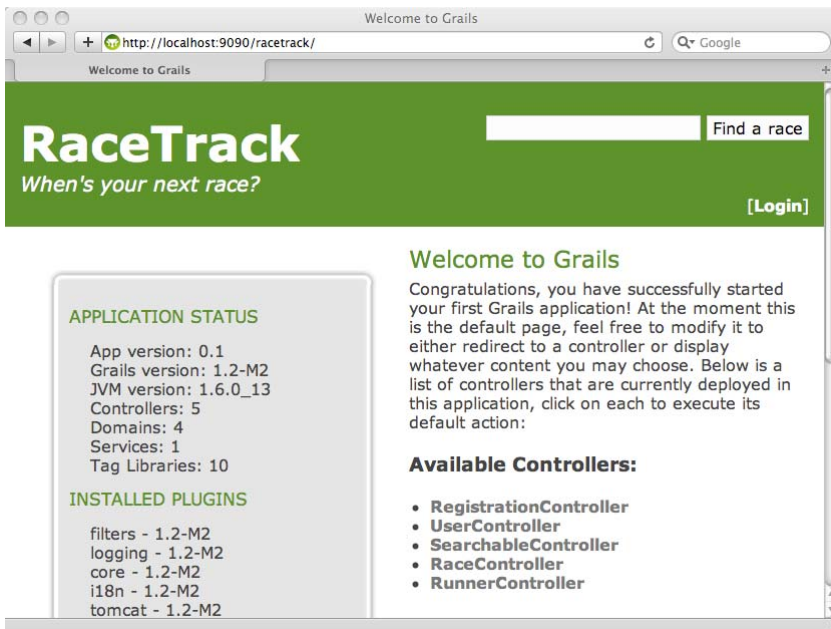
<div id="header">
  <p>
    <a class="header-main" href="${resource(dir: '')}">RaceTrack</a>
  </p>

  <p class="header-sub">When's your next race?</p>
  <div id="loginHeader">
    <g:loginControl />
  </div>
</div>
```

Add a bit of CSS to the bottom of `web-app/css/main.css` to position it correctly on the screen:

```
#search {
  float: right;
  margin: 2em 1em;
}
```

Now, visit <http://localhost:9090/racetrack> to see your work so far:



Well, the front end looks pretty good. But the back end is still as ugly as ever. Let's create a landing page for the search results. Name it `grails-app/views/race/list.gsp` so that we can use it for both the regular list view as well as search results.

```
<html>
  <head>
    <meta http-equiv="Content-Type"
      content="text/html; charset=UTF-8"/>
    <meta name="layout" content="main" />
    <title>RaceTrack</title>
  </head>
  <body>
    <div class="body">
      <g:if test="${flash.message}">
        <div class="message">${flash.message}</div>
      </g:if>

      <g:each in="${raceInstanceList}"
        status="i" var="raceInstance">
        <div class="race">
          <h2>${raceInstance.name}</h2>
          <p class="race-details">
            <span class="question">When?</span>
            <span class="answer">
              ${raceInstance.startDate.
                format("EEEE, MMMM d, yyyy")}</span>
          </p>
          <p class="race-details">
            <span class="question">Where?</span>
            <span class="answer">
              ${raceInstance.city}, ${raceInstance.state}</span>
          </p>
          <p class="race-details">
            <span class="question">How Long?</span>
            <span class="answer">
              <g:formatNumber
                number="${raceInstance.distance}"
                format="0 K" /></span>
          </p>
          <p class="race-details">
            <span class="question">How Much?</span>
            <span class="answer">
              <g:formatNumber
                number="${raceInstance.cost}"
                format="\$###,##0" /></span>
          </p>
        </div>
      </g:each>

      <div class="paginateButtons">
        <g:paginate total="${raceInstanceTotal}" />
      </div>
    </div>
  </body>
</html>
```

Add a bit more CSS to `web-app/css/main.css` help out with the look and feel:

```
.race {
  padding: 1em;
}

.race-details {
  padding-left: 2em;
}

.question {
  font-style: italic;
  font-weight: bold;
}
```

Next, adjust the code in `RaceController.search`:

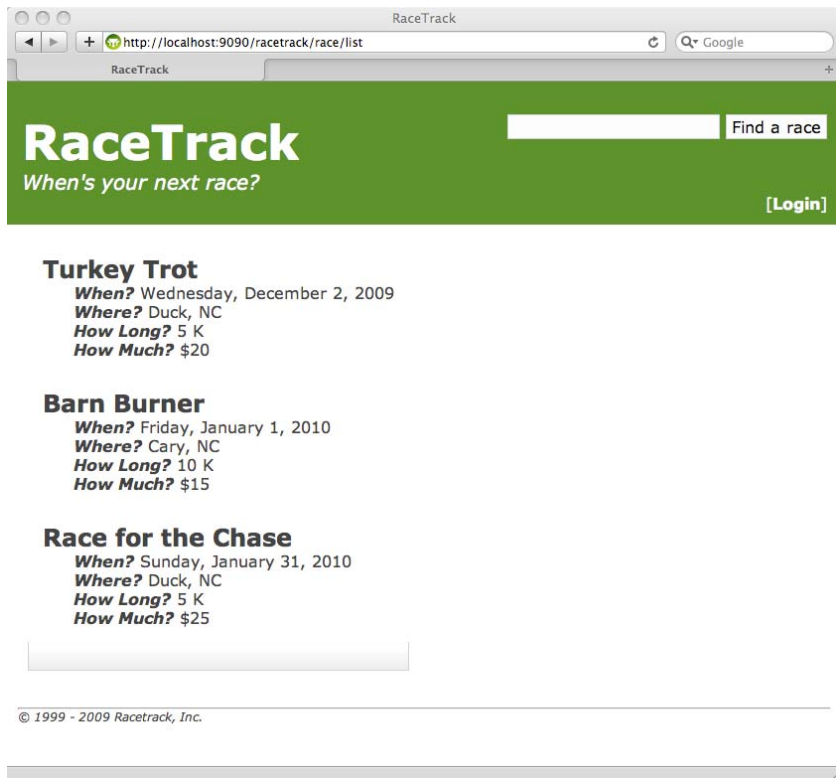
```
class RaceController {
  def scaffold = Race

  def search = {
    flash.message = "Search results for: ${params.q}"
    def resultsMap = Race.search(params.q, params)
    render(view: 'list',
           model: [
             raceInstanceList: resultsMap.results,
             raceInstanceTotal: Race.countHits(params.q)
           ])
  }
}
```

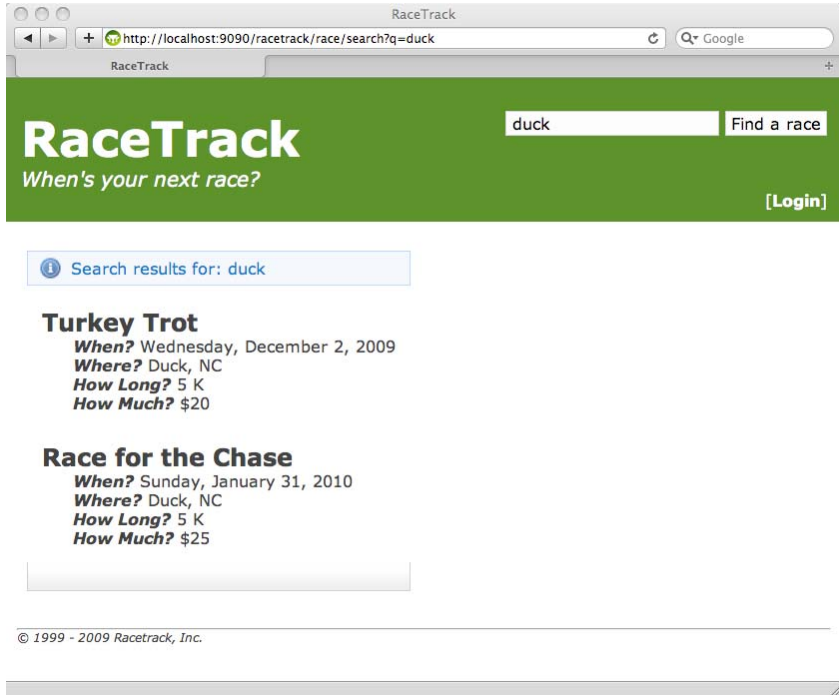
And finally, add a few more races to `grails-app/conf/BootStrap.groovy` to make things more interesting.

```
def burner = new Race(  
    name:"Barn Burner",  
    startDate:(new Date() + 120),  
    city:"Cary",  
    state:"NC",  
    distance:10.0,  
    cost:15.0,  
    maxRunners:350  
)  
burner.save()  
if(burner.hasErrors()){  
    println burner.errors  
}  
  
def chase = new Race(  
    name:"Race for the Chase",  
    startDate:(new Date() + 150),  
    city:"Duck",  
    state:"NC",  
    distance:5.0,  
    cost:25.0,  
    maxRunners:350  
)  
chase.save()  
if(chase.hasErrors()){  
    println chase.errors  
}
```

Let's see how we did. Take a look at <http://localhost:9090/racetrack/race> in your web browser.



And for the big test, search for something and view the results:



Code reuse at its finest, eh?

Our new list/search page looks so good, I think that I'd like to use it for my new home page instead of the "Welcome to Grails" boilerplate `index.gsp`. But I still want to keep the boilerplate page around – it'll make a decent admin screen. Let's put the finishing touches on our RaceTrack application and call it a day.

## Changing the Home Page with UrlMappings

We've spent quite a bit of time touting the benefits of convention over configuration. But rules are meant to be broken, and that includes the way that URLs get mapped to the underlying Grails Controllers. Luckily, `grails-app/conf/UrlMappings.groovy` lets us tweak things to our heart's content.

```
class UrlMappings {
    static mappings = {
        "/*controller/*action?/*id?" {
            constraints { // apply constraints here
            }
        }

        "/*"(view:"/index")
        "500"(view:'/error')
    }
}
```

As you can probably see, the first mapping is what maps <http://localhost:9090/racetrack/race/show/2> to `params.controller`, `params.action`, and `params.id`. But the second mapping is what we are here for. It is what maps the trailing slash in <http://localhost:9090/racetrack/> to `index.gsp`. Change that mapping to point to the `RaceController` `list` action, and you have a brand new home page:

```
class UrlMappings {
    static mappings = {
        "/*controller/*action?/*id?" {
            constraints { // apply constraints here
            }
        }

        "/*"(controller:"race", action:"list")
        "500"(view:'/error')
    }
}
```



With that done, let's repurpose the existing `index.gsp` page for an admin console. Create `grails-app/controllers/AdminController.groovy`:

```
class AdminController{
    def beforeInterceptor = [action:this.&auth]

    def auth() {
        if(!session.user) {
            redirect(controller:"user", action:"login")
            return false
        }

        if(!session.user.admin){
            flash.message = "Tsk tsk-admins only"
            redirect(controller:"race", action:"list")
            return false
        }
    }

    def index = {}
}
```

Notice the lowly empty `index` action at the bottom of the file? Move `grails-app/views/index.gsp` to `grails-app/views/admin/index.gsp`, and you have a perfectly good admin console. What's especially nice about this page is, as you have seen throughout the book, it will automatically add links for each new controller as you add them. There is nothing better than a zero-effort admin console:

```
<ul>
  <g:each var="c"
    in="${grailsApplication.controllerClasses}">
    <li class="controller">
      <g:link controller="${c.logicalPropertyName}">
        ${c.fullName}
      </g:link>
    </li>
  </g:each>
</ul>
```

You might even want to go back to `UserController` and make this page the default page for admins when they authenticate:

```
class UserController{
  // ...
  def authenticate = {
    def user = User.findByLoginAndPassword(
      params.login, params.password.encodeAsSHA())
    if(user){
      session.user = user
      flash.message = "Hello ${user.login}!"
      if(user.admin){
        redirect(controller:"admin", action:"index")
      } else{
        redirect(controller:"race", action:"list")
      }
    } else{
      flash.message
        = "Sorry, ${params.login}. Please try again."
      redirect(action:"login")
    }
  }
}
```

## Production Deployment Checklist

---

We are just about ready to generate a WAR file and deploy it to Tomcat, JBoss, GlassFish, or the application server of your choice. But before you do, there are a couple of things you should double-check.

The most important thing to verify is your database connections in `grails-app/conf/DataSource.groovy`. Be sure to point Grails to the correct database (preferably *not* the same one you use for test and development modes). If you are deploying to an application server that offers database connections via JNDI, you can replace the individual `username`, `password`, and `url` variables with a single `jndiName` variable:

```
production{
  dataSource {
    jndiName = "java:comp/env/jdbc/myDataSource "
  }
}
```

Next, check `grails-app/conf/BootStrap.groovy` for any production-specific settings. This file is typically used for development mode only, but you can always seed your production application with admin accounts and lookup data.

```
import grails.util.GrailsUtil

class BootStrap {
    def init = { servletContext ->
        switch(GrailsUtil.environment){
            case "development":
                // ...
                break

            case "production":
                // optionally seed production data here
                break
        }
    }
    def destroy = { }
}
```

Next, there is an `environments` configuration block in `grails-app/conf/Config.groovy`. You can set the server URL here, as well as any other system-wide values you'd like.

```
// set per-environment serverURL stem for creating
absolute links
environments {
    production {
        grails.serverURL = "http://www.changeme.com"
        the.word.is = "bird"
    }
}
```

If you need to access these variables in your Controllers or TagLibs, you can simply call `grailsApplication.config.grails.serverURL` or `grailsApplication.config.the.word.is`. In other classes, you need to explicitly import the `ConfigurationHolder` convenience class.

```
import org.codehaus.groovy.grails.commons.*

def grailsApplication = ConfigurationHolder
println grailsApplication.config.grails.serverURL
```

Next, adjust the `app.name` and `app.version` values in `application.properties`. These values are used to name the WAR file.

```
#utf-8
#Fri Jul 03 08:57:11 MDT 2009
app.version=0.1
app.servlet.version=2.4
app.grails.version=1.2.0
plugins.searchable=0.5.5
plugins.hibernate=1.2.0
plugins.tomcat=1.2.0
app.name=racetrack
```

And finally, you can put any generic JEE configuration settings in `src/templates/war/web.xml`. (If this file doesn't exist, type `grails install-templates` to generate it.)

If everything looks OK, then type `grails clean` to make sure that there aren't any lingering development-mode artifacts hanging around.

```
$ grails clean
...
[delete] Deleting directory
/src/racetrack/web-app/plugins
[delete] Deleting directory
/Users/sdavis/.grails/1.2.0/projects/racetrack/classes
[delete] Deleting directory
/Users/sdavis/.grails/1.2.0/projects/racetrack/resources
[delete] Deleting directory
/Users/sdavis/.grails/1.2.0/projects/racetrack/test-
classes
```

Once that's done, type `grails war`. You should end up with a genuine JEE WAR file, suitable for deploying to any Java application server.

```
$ grails war
...
Done creating WAR
/src/racetrack/racetrack-0.1.war
```

No additional server configuration is necessary to deploy your WAR – it includes the Groovy JAR, the Grails supporting JARs, and everything else that you added to the `lib` directory. Type `jar tvf racetrack-0.1.war` to see everything that is included in the WAR file.

### Free Online Version

Support this work, buy the print copy:

<http://www.infoq.com/minibooks/grails-getting-started>

## Conclusion

Well, I hope that you've enjoyed your time with Grails. Now that you know the basics, the only thing left to do is get experienced with it. As you've seen, you can get a simple application up and running in hours – not days, weeks, or months.

Type `grails stats` one last time.

```
$ grails stats
```

Name	Files	LOC
Controllers	5	164
Domain Classes	4	83
Tag Libraries	2	19
Unit Tests	12	187
Integration Tests	1	19
Totals	24	472

Wow – well under 500 lines of code, and you have reasonably sophisticated application to show for it. One that includes testing, security, and much more. This kind of speed and efficiency should motivate you to finally write that application that's been bugging you. You know the one I'm talking about – the “if I only had the time” project that's been sitting around for years.

If you work for a company, what administrivia can you automate with Grails? Perhaps a library application that lists the computer books your department owns and who has them checked out. Or maybe a system to track computer hardware inventory in your organization – manufacturers, models, serial numbers, etc.

If you are playing around with Grails at home, why not write an application to manage your DVD or music collection? You can build something to track your ever-present to-do list, your progress on your weight-loss plan, or even a grocery-list helper.

If you belong to a community group (like a Java User Group), you can put together a website that tracks meetings, presentations, and so on. If your children play sports, you can build an application to track practices, games, and away trips.

But don't be fooled into thinking that the simplicity of Grails means that it is only applicable for "toy" applications. There is an ever-growing, diverse set of companies that are using Grails for their public-facing websites. From Wired in the US (<http://wired.com/reviews>) to SkyTV in the UK (<http://sky.com/>) to Taco Bell in Canada (<http://tacobell.ca/>), new Grails websites are popping up all over the place. And for every public example of a Grails application, there are countless others that are humming along happily behind the firewall of major corporations everywhere.

For further learning, there are a number of excellent resources out there:

I think that you'll be impressed with the quality of the online Grails documentation:

<http://grails.org/Documentation>

I write an ongoing (free) article series for IBM DeveloperWorks called *Mastering Grails*:

<http://tinyurl.com/mastering-grails>

If you enjoy *Mastering Grails*, chances are good that you'll enjoy another series I write for IBM DeveloperWorks as well – *Practically Groovy*:

<http://tinyurl.com/practically-groovy>

My book *Groovy Recipes: Greasing the Wheels of Java* is a great way to see what Groovy can do outside of the context of web development:

<http://pragprog.com/titles/sdgrvr/groovy-recipes>

As I speak at software conferences internationally, I try to chat up Groovy and Grails luminaries and video tape the results. Watch interviews with Graeme Rocher and other folks from the Groovy/Grails community:

<http://thirstyhead.blip.tv/>

I kicked this book off saying, "Grails is an open-source web application framework that's all about getting things done." Hopefully you agree. Enjoy Grails!

## About the Authors

---

**Scott Davis** (email: [scott@thirstyhead.com](mailto:scott@thirstyhead.com)) is the founder of ThirstyHead.com, a training company that specializes in Groovy and Grails training.

Scott published one of the first public websites implemented in Grails in 2006 and has been actively working with the technology ever since. Author of the book *Groovy Recipes: Greasing the Wheels of Java* and two ongoing IBM developerWorks article series (*Mastering Grails* and in 2009, *Practically Groovy*), Scott writes extensively about how Groovy and Grails are the future of Java development.

Scott teaches public and private classes on Groovy and Grails for start-ups and Fortune 100 companies. He is the co-founder of the Groovy/Grails Experience conference and is a regular presenter on the international technical conference circuit (including QCon, No Fluff Just Stuff, JavaOne, OSCON, and TheServerSide). In 2008, Scott was voted the top Rock Star at JavaOne for his talk "Groovy, the Red Pill: How to blow the mind of a buttoned-down Java developer".

**Jason Rudolph** is a Partner at Relevance, a leading consultancy and training organization specializing in Ruby, Rails, Groovy, and Grails and integrating them into enterprise environments. Jason has more than a decade of experience in developing software solutions for domestic and international clients of all sizes, including start-ups, Dow 30 companies, and government organizations.

Jason speaks frequently at software conferences and user groups, and also he's been known to write about software development from time to time. Jason was an early committer to Grails and contributes regularly to the open source community. More recently, Jason can be found working on Tarantula, Blue Ridge, and other Ruby and Rails projects dedicated to testing and improving code quality.

Jason holds a degree in Computer Science from the University of Virginia. He currently lives in Raleigh, NC with his wife (who can take a

functional web app and make it actually look good) and his dog (who is hard at work on her upcoming book, *Getting Started with Squirrels*).

You can find Jason online at <http://jasonrudolph.com>





