Mastering Grails: Build your first Grails application

See how much power is packed into this tiny Web framework package

Skill Level: Introductory

Scott Davis (scott@aboutgroovy.com)
Editor in Chief
AboutGroovy.com

15 Jan 2008

Java™ programmers needn't abandon their favorite language and existing development infrastructure to adopt a modern Web development framework. In the first installment of his new monthly series *Mastering Grails*, Java expert Scott Davis introduces Grails and demonstrates how to build your first Grails application.

Allow me to introduce you to Grails by way of another open source Web development framework: Ruby on Rails. When Rails was released, it captivated developers. Rails's scaffolding capabilities let you bootstrap a new project in a fraction of the time it previously took. The *convention over configuration* idea underlying Rails means that your application auto-wires itself together based on common-sense naming schemes rather than tedious and error-prone XML configuration files. Ruby's metaprogramming capabilities let objects magically inherit the methods and fields they need at run time without cluttering up the source code.

Rails was (and still is) worthy of the accolades and adoration it has received, but it presents Java developers with difficult choices. Do you abandon the familiar Java platform for the promise of a new one? And what do you do with your existing Java code, your existing production servers, and your staff of seasoned Java developers?

About this series
Grails is a modern Web development framework that mixes familiar Java technologies like Spring and Hibernate with contemporary practices like convention over configuration. Written in Groovy, Grails gives you seamless integration with your legacy Java code

while adding the flexibility and dynamism of a scripting language. After you learn Grails, you'll never look at Web development the same way again.

Enter Grails. Grails gives you the development experience of Rails while being firmly grounded in proven Java technologies. But Grails isn't just a simple "me too" port of Rails to the Java platform. Grails takes the lessons learned from Rails and mixes them with the sensibilities of modern Java development. Think *inspired by*, not *translated from*.

To inaugurate the *Mastering Grails* series, this article introduces you to the Grails framework, shows you how to install it, and walks you through building your first Grails application: a trip planner that you'll continue working on in subsequent installments in this series.

The power of Groovy

Just as Rails is deeply tied to the Ruby programming language, Grails wouldn't exist without the power of Groovy (see Resources). Groovy is a dynamic language that runs on the JVM and seamlessly integrates with the Java language. If you've read the long-running *Practically Groovy* series on developerWorks, you're already familiar with the language's power. If you haven't, don't worry — as you learn Grails, you'll pick up plenty of Groovy along the way. It shouldn't be hard because Groovy was expressly designed to appeal to Java developers.

For example, Groovy allows you to reduce dramatically the amount of Java code that you would normally write. You no longer need to write getters and setters for your fields, because Groovy provides them for you automatically. No more writing `for Iterator i = list.iterator()` to loop through a list of items; `list.each` does the same thing more concisely and more expressively. Simply put, Groovy is what the Java language would look like had it been written in the twenty-first century.

Groovy wouldn't be nearly as appealing to Java developers if it forced them to rewrite entire applications from the ground up to take advantage of it. Happily, Groovy seamlessly integrates with your existing codebase. Groovy doesn't replace the Java language — it's an enhancement. You can quickly pick up Groovy because at the end of the day, Groovy code *is* Java code. The two languages are so compatible that you can rename a working .java file to a .groovy file — for example, change Person.java to Person.groovy — to have a perfectly valid (and executable) Groovy file (albeit one that doesn't take advantage of any of the syntactic sugar Groovy provides).

This deep level of compatibility between Groovy and the Java language means that Grails doesn't need to reinvent the wheel when it comes to the key technologies

used behind the scenes. Instead, it allows you to look at familiar Java libraries through Groovy-colored glasses. JUnit `TestCases` are wrapped in Groovy and presented as `GroovyTestCases`. Grails offers a new twist on Ant builds with GANT, a pure-Groovy implementation of Ant. Grails wraps Hibernate in a thin Groovy facade and calls it GORM — the Grails Object/Relational Mapper. These are just three examples of how Grails allows you to leverage all of your existing Java experience while taking advantage of modern Web development practices.

But to appreciate Grails fully, you have to experience it first-hand. It's time to install Grails and create your first Web application.

Installing Grails

Everything you need to run a Grails application is included in a single ZIP file. All the dependent libraries — Groovy, Spring, and Hibernate, to name just a few — are already in place and ready for use. To install Grails:

1. Download and unzip grails.zip from the Grails site (see Resources).
2. Create a `GRAILS_HOME` environment variable.
3. Add $GRAILS_HOME/bin to the `PATH`.

Okay, you *do* need to have a JDK installed. (Grails is good, but it's not *that* good.) Grails 1.0 runs on Java 1.4, 1.5, and 1.6. If you don't know which version you have installed, type `java -version` at a command prompt. If necessary, download and install a Grails-compatible version of the JDK (see Resources).

When you've finished the installation steps, type `grails -version` to check your work. If you are greeted with this friendly banner, you know that everything is configured correctly:

```
Welcome to Grails 1.0 - http://grails.org/
Licensed under Apache Standard License 2.0
Grails home is set to: /opt/grails
```

Web server and database included

Using the freebies
For this article's application, you'll use the Web server and database that Grails provides for free. In a future installment, I'll give you the step-by-step instructions for running Grails against your own servers. In the meantime, don't be shy about wandering over to grails.org and browsing through the excellent online documentation (see Resources).

Interestingly, you don't need a separate Web server installed to run Grails applications. Grails ships with its own embedded Jetty servlet container. You type `grails run-app` to get an application up and running in the Jetty container (see Resources) without needing to hop through the usual deployment hoops. Running a Grails application on your existing production server is no problem either. Typing `grails war` creates a standard file that you can deploy to Tomcat, JBoss, Geronimo, WebSphere®, or any other Java EE 2.4-compliant servlet container.

You don't need to have a separate database installed, either. Grails ships with HSQLDB (see Resources), a pure-Java database. Having a database ready to use out of the gate gives you an instant boost in productivity. Using another database, such as MySQL, PostgreSQL, Oracle Database, or DB2 is simple thanks to Hibernate and GORM. If you have a JDBC driver JAR and the usual connection settings handy, one change to DataSource.groovy has you talking to your own database in no time.

Writing your first Grails application

I do a lot of traveling — at least 40 trips a year. I've found that calendars do a great job of telling me *when* I need to be somewhere but don't do as well showing me *where* that place is. Online maps have the opposite problem: they're great with the where, not so great with the when. So in this and the next couple of articles in this series, you'll put together a custom Grails application that will help with both the when and the where of planning a trip.

> **Spoiler alert**
> Do you see future articles in this series talking about using Grails in conjunction with Google Calendar and Google Maps? I sure do....

To begin, start in a clean directory and type `grails create-app trip-planner`. After a flurry of activity, you'll see a directory named trip-planner. Like Maven, Rails, and AppFuse, Grails scaffolds out a standard directory structure for you. If you feel hopelessly constrained by this limitation and cannot work with a framework unless you can meticulously design your own custom directory tree, you won't have much fun working with Grails. The *convention* part of convention over configuration lets you sit down with any Grails application and know immediately which bits are stored in what bucket.

Change to the trip-planner directory and `type grails create-domain-class Trip`. If all goes well, you'll have two new files waiting for you: grails-app/domain/Trip.groovy and grails-app/test/integration/TripTests.groovy. I'll cover testing in a later article. For right now, I'll focus on the domain class, which starts out looking like Listing 1:

Listing 1. Grails-generated domain class

```
class Trip{

}
```

Not much to look at, eh? Well, let's fix that. Add fields to `Trip` as shown in Listing 2:

Listing 2. Trip class with added fields

```
class Trip {
  String name
  String city
  Date startDate
  Date endDate
  String purpose
  String notes
}
```

As I mentioned earlier, you don't need to worry about creating the getters and setters: Groovy dynamically generates them for you. `Trip` also gets lots of new and useful dynamic methods whose names are handily self-explanatory:

- `Trip.save()` *saves* the data to the `Trip` table in the HSQLDB database.
- `Trip.delete()` *deletes* the data from the `Trip table`.
- `Trip.list()` returns a list of `Trips`.
- `Trip.get()` returns a single `Trip`.

All of these methods and more are simply there when you need them. Notice that `Trip` doesn't extend a parent class or implement a magic interface. Thanks to Groovy's metaprogramming capabilities, those methods simply appear in the proper place on the proper classes. (Only classes in the grails-app/domain directory get these persistence-related methods.)

Building the controller and views

Creating the domain class is only half the battle. Every model needs a good controller and some views to even things out. (I'm assuming you're familiar with the Model-View-Controller pattern; see Resources.) Type `grails generate-all Trip` to build out a grails-app/controllers/TripController.groovy class and a matching set of Groovy Server Pages (GSPs) in grails-app/views/Trip. For every `list` action in a controller, there's a corresponding list.gsp file. The `create` action has a create.gsp. Here you see the benefits of convention over configuration in action: no XML files are required to match these elements up. Every domain class pairs up with a controller based on the name. Every action in a controller pairs up with a view

based on the name. You can bypass this name-based configuration if you like, but most of the time you just follow the convention, and your application works out of the box.

Take a look at grails-app/controller/TripController.groovy, shown in Listing 3:

Listing 3. TripController

```
class TripController {
    ...
    def list = {
        if(!params.max) params.max = 10
        [ tripList: Trip.list( params ) ]
    }
    ...
}
```

The first thing Java developers tend to notice is how much is accomplished with so few lines of code. Take the `list` action, for example. The important thing going on in the method is the last line. Grails is returning a hashmap with a single element named `tripList`. (The last line of a Groovy method is an implicit return statement. You can also manually type in the word `return` if you'd like.) The `tripList` element is an `ArrayList` of `Trip` objects pulled from the database by the `Trip.list()` method. Normally this method returns all the records from the table. The line immediately above it says, "Hey, if anyone passes in a max parameter in the URL, use it to limit the number of `Trips` returned. If not, limit the number of `Trips` to 10." The URL http://localhost:8080/trip-planner/trip/list invokes this action. For example, http://localhost:8080/trip-planner/trip/list?max=3 displays 3 trips instead of the usual 10. If you have more trips to display, Grails creates previous and next pagination links automatically.

So, where does this hashmap get used? Take a look at grails-app/views/list.gsp, shown in Listing 4:

Listing 4. list.gsp

```
<g:each in="${tripList}" status="i" var="trip">
  <tr class="${(i % 2) == 0 ? 'odd' : 'even'}">
    <td>
      <g:link action="show" id="${trip.id}">${trip.id?.encodeAsHTML()}</g:link>
    </td>
  </tr>
</g:each>
```

list.gsp is mostly plain old HTML with a smattering of GroovyTagLibs. Anything prefixed with a `g:` is a GroovyTag. In Listing 4, `<g:each>` walks through each `Trip` in the `tripList ArrayList` and builds out a well-formed HTML table.

Understanding controllers boils down to the three *R*s: *return*, *redirect*, and *render*.

Several of the actions take advantage of the implicit return statement to return data to a GSP page named the same. Some actions redirect you. For example, `index` is the action that gets called if you don't specify one in the URL:

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

In this case, `TripController` redirects you to the `list` action, passing along all of the parameters (or `QueryString`) in the `params` hashmap.

Finally, the `save` action (see Listing 5) doesn't have a corresponding save.gsp page. It redirects you to the `show` action page if the record gets saved to the database without error. Otherwise it renders the create.gsp page, where you can see the errors and try again.

Listing 5. The save action

```
def save = {
  def trip = new Trip(params)
  if(!trip.hasErrors() && trip.save()) {
    flash.message = "Trip ${trip.id} created"
    redirect(action:show,id:trip.id)
  }
  else {
    render(view:'create',model:[trip:trip])
  }
}
```

But rather than talk more about how Grails works, let's see it in action.

The application in action

Type `grails run-app` at the command line. After a flurry of Log4j messages fly past on the console, you should be rewarded with a message that says:

```
Server running. Browse to http://localhost:8080/trip-planner
```

If you already have a server running on port 8080, you'll be scolded with a core dump that ends with:

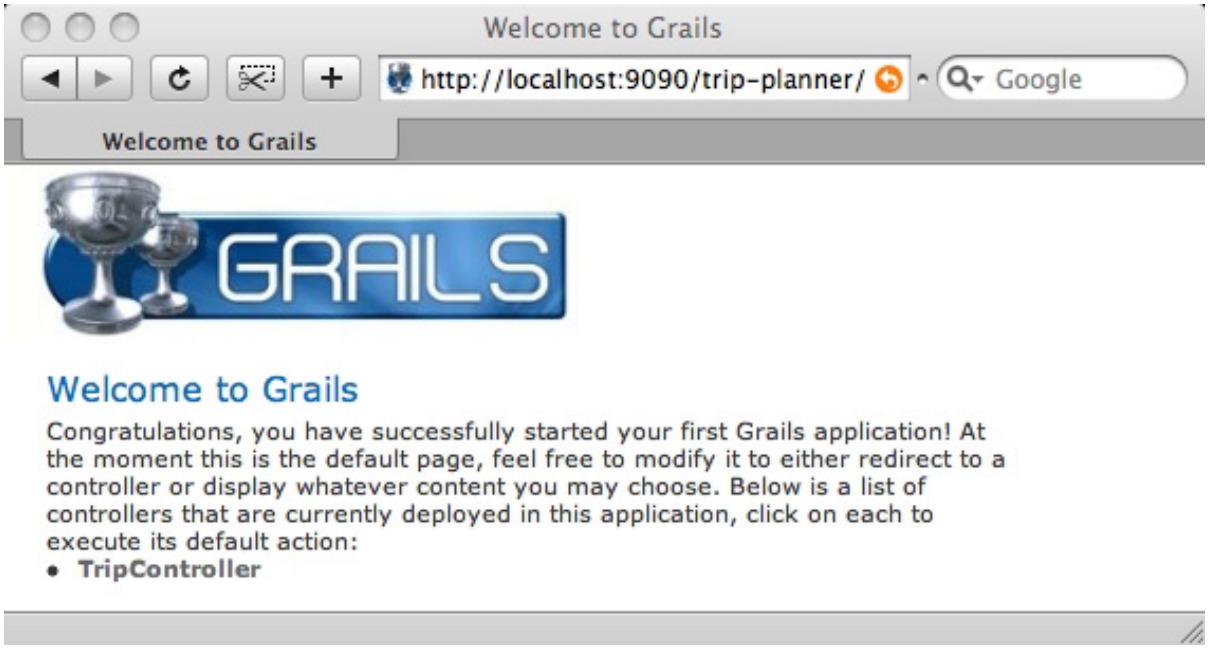
```
Server failed to start: java.net.BindException: Address already in use
```

You can easily change the port that Jetty runs on in two ways. You can make an ad hoc change by typing `grails -Dserver.port=9090 run-app`. To make that change permanent, look in $GRAILS_HOME/scripts/Init.groovy for the line that begins with `serverPort` and change the value:

```
serverPort = System.getProperty('server.port') ?
             System.getProperty('server.port').toInteger() : 9090
```

Once you have Grails running on the port of your choice, type the URL into your Web browser. You should see a welcome screen listing out all of your controllers, as shown in Figure 1:

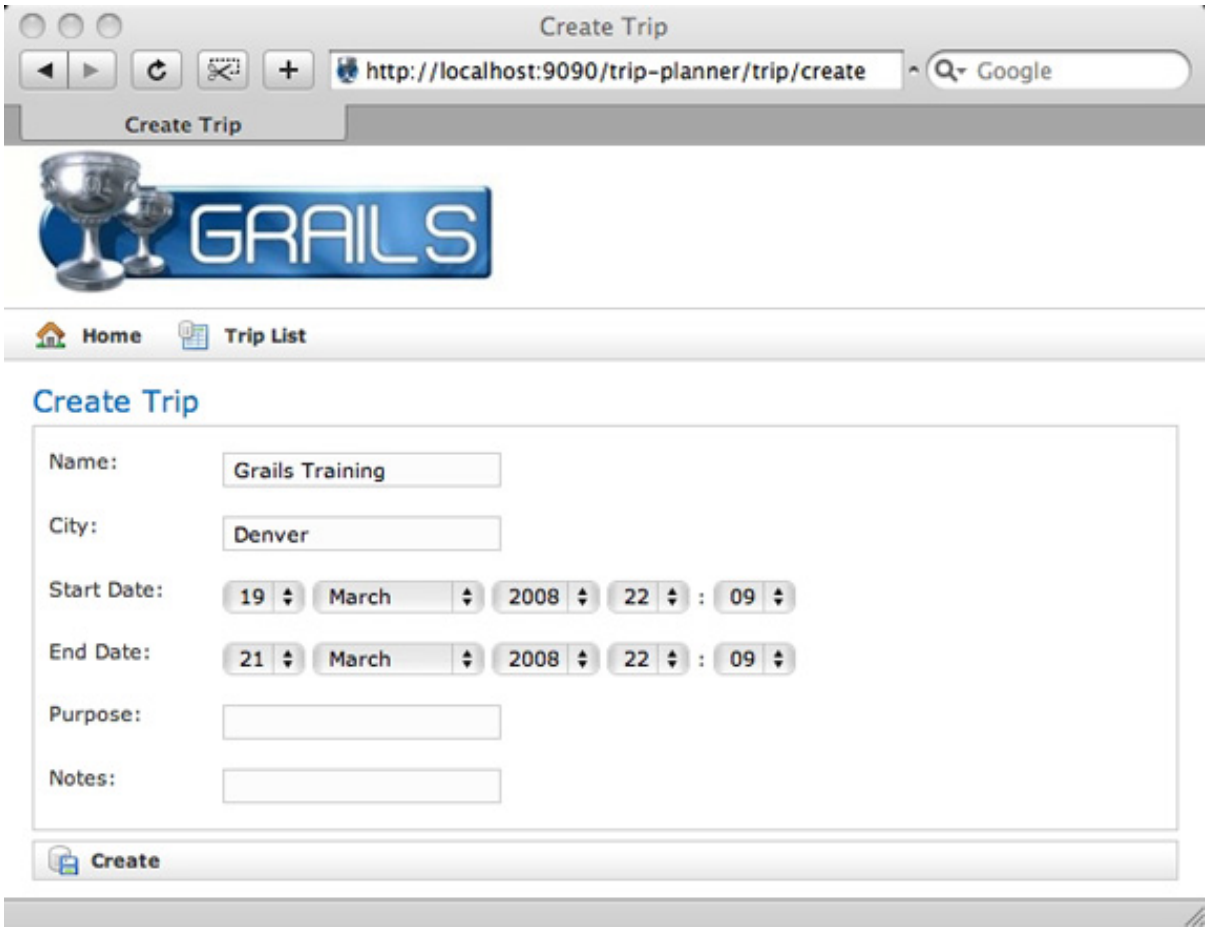
Figure 1. Grails application welcome screen



Click on the **TripController** link. You have a full CRUD (create, read, update, delete) application at your fingertips to play around with.

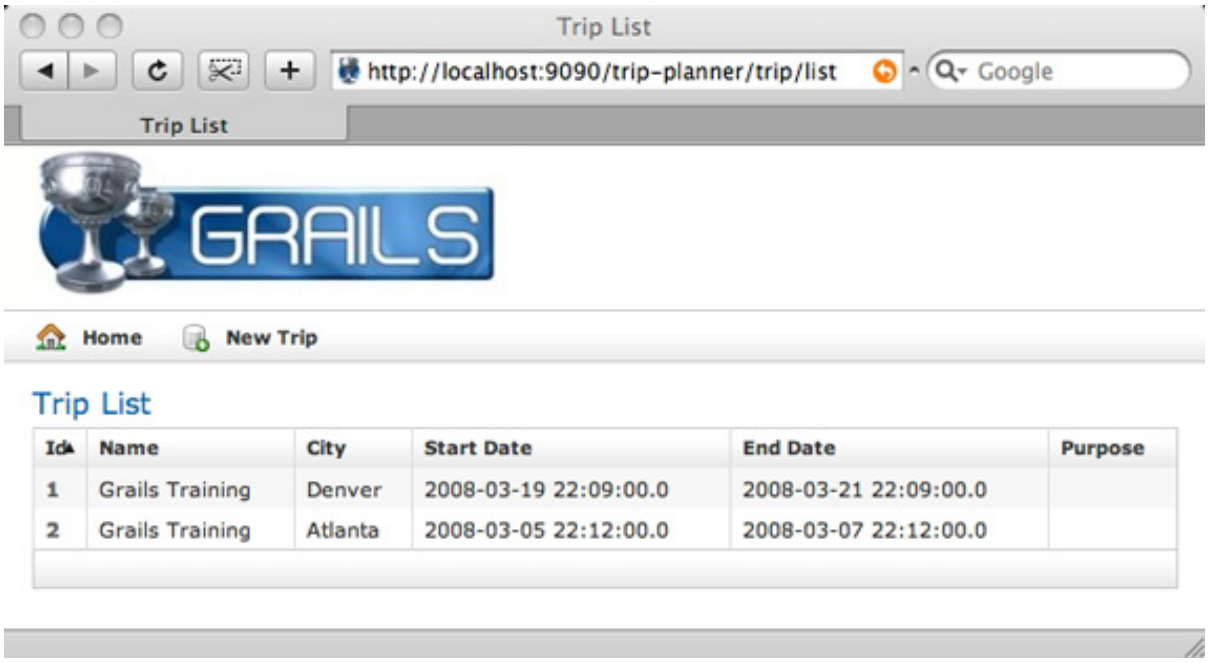
Create new trips using the page shown in Figure 2:

Figure 2. Create Trip page



Edit your trips using the page shown in Figure 3:

Figure 3. The Trip List page



And how long did it take you to get the application up and running? How many lines of code did it take? Here's how you find out:

1. Press Ctrl-C to shut down Grails.
2. Type `grails stats`.

You'll see this output on the screen:

```
+----------------------+-------+-------+
| Name                 | Files |  LOC  |
+----------------------+-------+-------+
| Controllers          |     1 |    66 |
| Domain Classes       |     1 |     8 |
| Integration Tests    |     1 |     4 |
+----------------------+-------+-------+
| Totals               |     3 |    78 |
+----------------------+-------+-------+
```

It took fewer than 100 lines of code to implement all of the application's functionality. Not bad at all. But allow me to show you one more trick before I end for the day.

Generating the controller and views is a great learning exercise, and having physical files on disk helps illustrate how everything is wired together. But do me a favor: delete the contents of the `TripController` class and replace it with:

```
class TripController{
  def scaffold = Trip
}
```

This single line of code tells Grails to do everything that it was doing with the previous controller, only to generate all of those `list`, `save`, and `edit` actions in memory dynamically at run time. Three lines of code instead of 66 produce all of the same behavior.

Type `grails run-app` again. Yes — all your data went away. Don't sweat it. Press Ctrl-C to shut Grails down. This time, type `grails prod run-app`. You're now running in production mode, which means that your data is saved between server restarts. Click your way through to the `TripController` and save a few more records. You shouldn't see any difference in the application's behavior. Knowing that everything you see in the browser is being driven by 15 lines of code gives you an idea of the power of Grails.

Conclusion

I hope you've enjoyed your first taste of Grails. An amazing amount of power is packed into a tiny package, and you've only scratched the surface. Installing the framework involved little more than unzipping a file. Creating an application from scratch involved typing a couple of commands. I hope this whirlwind tour whets your appetite for more Grails. It certainly sets the stage for you to expand on this example and take it in all sorts of new and interesting directions.

In next month's installment, you'll spend some quality time with GORM. You'll save your data out to a MySQL database, put some data validation in place, and set up a one-to-many relationship. Without adding many lines of code, you'll increase the trip-planner application's capabilities significantly.

Until then, have fun playing around with Groovy and Grails. You'll never look at Web development the same way again.

Resources

Learn

- Grails: Visit the Grails Web site.
- *Practically Groovy:* This developerWorks series explores the practical uses of Groovy and teaches you when and how to apply them.
- Groovy: Learn more about Groovy at the project Web site.
- AboutGroovy.com: Keep up with the latest Groovy news and article links.
- "What's the secret sauce in Ruby on Rails?" (Bruce Tate, developerWorks, May 2006): This article in Bruce Tate's *Crossing borders* series on developerWorks explains why Rails has made such a big splash.
- HSQLDB and Jetty: Explore the pure-Java database and servlet container that ship with Grails.
- Model-View-Controller: Grails follows this popular architectural pattern.
- Browse the technology bookstore for books on these and other technical topics.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- Grails: Download the latest Grails release.
- Java SE: Download a Grails-compatible JDK.
- IBM developer kits for the Java platform: If you'd like to use Grails with the IBM developer kit, be sure to read this FAQ first for tips.

Discuss

- Check out developerWorks blogs and get involved in the developerWorks community.

About the author

Scott Davis
Scott Davis is an internationally recognized author, speaker, and software developer. His books include *Groovy Recipes: Greasing the Wheels of Java*, *GIS for Web Developers: Adding Where to Your Application*, *The Google Maps API*, and *JBoss At Work*.

Trademarks

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.
WebSphere is a trademark of IBM Corporation in the United States, other countries, or both.