Mastering Grails: Changing the view with Groovy Server Pages

Skill Level: Introductory

Scott Davis
Editor in Chief
AboutGroovy.com

11 Mar 2008

Groovy Server Pages (GSP) puts the *Web* in the Grails Web framework. In the third installment of his *Mastering Grails* series, Scott Davis shows you the ins and outs of working with GSP. See how easy it is to use Grails TagLibs, mix together partial fragments of GSPs, and customize the default templates for the automatically generated (scaffolded) views.

The first two articles in this series introduced you to the basic building blocks of the Grails Web framework. I've told you — repeatedly — that Grails is based on the Model-View-Controller (MVC) architectural pattern (see Resources), and that Grails favors *convention over configuration* for binding the framework's parts together. Grails uses intuitively named files and directories to replace the older, more error-prone method of manually cataloguing these linkages in an external configuration file. For example, in the first article you saw that controllers have a `Controller` suffix and are stored in the grails-app/controller directory. In the second article, you learned that domain models can be found in the grails-app/domain directory.

This month, I'll round out the MVC triptych with a discussion of Grails views. Views (as you might guess) are stored in the grails-app/views directory. But there's much more to the view story than the intuitively obvious directory name. I'll talk about Groovy Server Pages (GSP) and give you pointers to many alternative view options. You'll learn about the standard Grails tag libraries (TagLibs) and find out how easy it is to create your own TagLib. You'll see how to fight the ongoing battle for DRYness (see Resources) by factoring common fragments of GSP code into their own partial templates. Finally, you'll learn how to tweak the default templates for scaffolded views, thereby balancing the convenience of automatically created views with the

desire to move beyond a Grails application's default look-and-feel.

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 give 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.

Viewing a Grails application

Grails uses GSP for the presentation tier. The *Groovy* in Groovy Server Pages not only identifies the underlying technology, but also the language you can use if you want to write a quick scriptlet or two. In that sense, it's much like Java™Server Pages (JSP) technology, which allows you to mix a bit of Java code into your Web pages, and RHTML (the core view technology for Ruby on Rails), which let you sneak a bit of Ruby in between your HTML tags.

Of course, scriptlets have long been frowned upon in the Java community. They lead to the lowest form of technology reuse — copy-and-paste — and other crimes of technological moral turpitude. (There's a huge disparity between *because you can* and *because you should*.) The *G* in GSP should remind fine, upstanding Java citizens of the implementation language and nothing more. Groovy TagLibs and partial templates give you a more sophisticated way of sharing code and behavior across Web pages.

GSPs are the foundation of Grails' page-centric MVC world view. The page is the basic unit of measure. The List page offers links to the Show page. The Show page allows you to click through to the Edit page, and so on. Whether you're a seasoned Struts developer or a recent Rails enthusiast, you're already familiar with this type of Web life cycle.

I mention this because a Cambrian explosion of non-page-centric view technologies has occurred in recent years (see Resources). Component-oriented Web frameworks such as JavaServer Faces (JSF) and Tapestry are gaining mindshare. The Ajax revolution has spawned numerous JavaScript-based solutions such as Dojo and the Yahoo! UI (YUI) libraries. Rich Internet Application (RIA) platforms such as Adobe Flash and the Google Web Toolkit (GWT) promise the convenience of Web deployment with a *richer*, desktop-like user experience. Luckily, Grails can handle all of these view technologies with ease.

The whole point of the MVC separation of concerns is that it lets you easily *skin* your Web application with any view you'd like. Grails' popular plug-in infrastructure means

that many GSP alternatives are just a `grails install-plugin` away. (See Resources for a link to the complete list of available plug-ins, or type `grails list-plugins` in a command prompt.) Many of these plug-ins are community-driven, the result of people wanting to use Grails with their presentation-tier technology of choice.

Although Grails has no native, automatic hooks for JSF, nothing precludes you from using the two together. A Grails application is a standard Java EE application, so you can put the proper JARs in the lib directory, add the expected settings in the WEB-INF/web.xml configuration file, and write the application as you normally would. A Grails application is deployed in a standard servlet container, so Grails supports JSPs just as well as GSPs. Grails plug-ins exist for Echo2 and Wicket (both component-oriented Web frameworks), so nothing is standing in the way of JSF or Tapestry plug-ins.

Similarly, the steps for adding Ajax frameworks such as Dojo and YUI to Grails are no different from what you would normally do: simply copy their JavaScript libraries to the web-app/js directory. Prototype and Scriptaculous are installed with Grails by default. The RichUI plug-in takes a best-of-breed approach, choosing UI widgets from a variety of Ajax libraries.

If you look through the list of plug-ins, you'll see support for such RIA clients as Flex, OpenLazlo, GWT, and ZK. Obviously, there's no shortage of alternate view solutions for Grails applications. But let's talk more about the native view technology that Grails supports out of the box — GSP.

GSP 101

You can spot a GSP page in a couple of ways. A file extension of .gsp is a dead giveaway, as is the copious use of tags that begin with `<g:`. As a matter of fact, a GSP page is nothing more than standard HTML with some Grails tags mixed in for dynamic content. Some of the alternative view technologies I mentioned in the preceding section are opaque abstraction layers that hide the details of the HTML, CSS, and JavaScript behind a layer of Java, ActionScript, or some other programming language. GSP is a thin Groovy facade over standard HTML, making it easy to *shell out* of the framework and use native Web technologies whenever you feel the need.

But you'll have a hard time finding GSPs in the Trip Planner application as it stands now. (The first two articles in this series get you started with building the Trip Planner. If you haven't followed along so far, now would be a good time to catch up.) You're currently using dynamic scaffolding for your views, so the trip-planner/grails-app/views directory is empty. Open grails-app/controller/TripController.groovy, shown in Listing 1, in a text editor to see the command used to enable dynamic scaffolding:

Listing 1. TripController class

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

The `def scaffold = Trip` line instructs Grails to generate the GSPs dynamically at run time. This is great for automatically keeping the views in sync as the domain model changes, but it doesn't give you much to look at while you are trying to learn the framework.

Type `grails generate-all Trip` in the root of the trip-planner directory. Answer `y` when it asks if you want to override the existing controller. (You can also answer `a` for *all* to override everything without being prompted repeatedly.) You should now see a full `TripController` class with closures named `create`, `edit`, `list`, and `show` (among others). You should also see a grails-app/views/trip directory with four GSPs: create.gsp, edit.gsp, list.gsp, and show.gsp.

Convention over configuration is at play here. When you visit http://localhost:9090/trip-planner/trip/list, you are asking the `TripController` to populate a list of `Trip` domain model objects and pass it on to the trip/list.gsp view. Take a look at TripController.groovy, shown in Listing 2, in a text editor once again:

Listing 2. Fully populated TripController class

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

This short closure retrieves 10 `Trip` records from the database, converts them to POGOs, and stores them in an `ArrayList` named `tripList`. The list.gsp page then iterates through the list, building an HTML table row by row.

The next section explores many popular Grails tags, including the `<g:each>` tag used to display each `Trip` in the Web page.

Grails tags

`<g:each>` is a commonly used Grails tag. It iterates over *each* item in a list. To see it in action, open grails-app/views/trip/list.gsp (shown in Listing 3) in a text editor:

Listing 3. The list.gsp view

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

The `status` attribute in the `<g:each>` tag is a simple counter field. (Notice this value is used on the next line in a ternary statement that sets the CSS style to either `even` or `odd`.) The `var` attribute allows you to name the variable used to hold the current item. If you change the name to `foo`, you need to change the later lines to `${foo.airline?.encodeAsHTML()}` and so on. (The `?.` operator is a Groovy way of avoiding `NullPointerExceptions`. It's shorthand for saying "call the `encodeAsHTML()` method only if `airline` is not null; otherwise just return an empty string.")

Another common Grails tag is `<g:link>`. As you might have already guessed, it builds an HTML `<a href>` link. Nothing is stopping you from using an `<a href>` tag directly, but this convenience tag accepts attributes such as `action`, `id`, and `controller`. If you'd like to get your hands on just the `href` value without the surrounding `anchor` tags, you can use `<g:createLink>` instead. At the top of list.gsp, you can see a third tag that returns a link: `<g:createLinkTo>`. This tag accepts `dir` and `file` attributes instead of logical `controller`, `action`, and `id` attributes. Listing 4 shows `link` and `createLinkTo` in action:

Listing 4. link vs. createLinkTo tags

```
<div class="nav">
  <span class="menuButton"><a class="home" href="${createLinkTo(dir:'')}">Home</a></span>
  <span class="menuButton"><link class="create" action="create">New Trip</g:link></span>
</div>
```

Notice in Listing 4 that you can call Grails tags in two different forms interchangeably — either as tags inside angle brackets or as method calls inside curly braces. The curly-brace notation (formally known as Expression Language or EL syntax) is better suited for times when the method call is embedded in another tag's attributes.

Just a few lines down in list.gsp, you can see another popular Grails tag in action: `<g:if>`, shown in Listing 5. In this case, it says "if the `flash.message` attribute is not null, display it."

Listing 5. <g:if> tag

```
<h1>Trip List</h1>
<if test="${flash.message}">
  <div class="message">${flash.message}</div>
</g:if>
```

As you browse through the generated views, you'll come across many other Grails tags in action. The `<g:paginate>` tag displays "previous" and "next" links if the database contains more `Trips` than the current 10 on display. The `<g:sortable>` tag makes the column headers clickable for sorting purposes. Nosing around the other GSP pages shows tags related to HTML forms, such as `<g:form>` and `<g:submit>`. The online Grails documentation lists all the available Grails tags and gives examples of their use (see Resources).

Custom tag libraries

Although the standard Grails tags are quite helpful, you'll eventually run into a situation where you want your own custom tags. Many seasoned Java developers (myself included) say publicly, "Yes, a custom TagLib is the appropriate architectural solution to use here," and then sneak off to write a scriptlet instead when they think that nobody is looking. Writing a custom JSP TagLib requires so much additional effort that scriptlets often win the day by virtue of being the path of least resistance. They aren't the right way to do things, but they are (unfortunately) the easy way.

Scriptlets — ugh. They are hacks in the truest sense of the word. They break HTML's tag-based paradigm and introduce raw code directly into your view. It's not even the code itself that's so bad; what's bad is the lack of encapsulation and reuse potential. The only way to reuse a scriptlet is by copy-and-paste. This leads to bugs, code bloat, and blatant DRY violations. And don't even get me started on a scriptlets' lack of testability.

That said, I must confess that I've written more than my fair share of JSP scriptlets when deadlines got tight. The JSP Standard Tag Library (JSTL) went a long way toward helping me get beyond my evil ways, but writing my own custom JSP tags was another issue altogether. By the time I wrote my custom JSP tag in Java code, compiled it, and messed around with getting the Tag Library Descriptor (TLD) in the right format and in the right place, I'd completely forgotten the reason I wrote the tag in the first place. As for writing tests to validate my new JSP tag — let's just say that my intentions were good.

In contrast, writing custom TagLibs in Grails is a breeze. The framework makes it easy to do the right thing, including writing tests. For example, I commonly need a boilerplate copyright notice at the bottom of Web pages. It should read *© 2002 - 2008, FakeCo Inc. All Rights Reserved.* Here's the thing: I'd like the second year always to be the current year. Listing 6 shows how you'd do this with a scriptlet:

Listing 6. Copyright notice with a scriptlet

```
<div id="copyright">
&copy; 2002 - ${Calendar.getInstance().get(Calendar.YEAR)},
    FakeCo Inc. All Rights Reserved.
</div>
```

Now that you know how to hack in the current year, next you'll create a custom tag that does the same thing. To start, type `grails create-tag-lib Date`. This creates two files: grails-app/taglib/DateTagLib.groovy (the TagLib) and grails-app/test/integration/DateTagLibTests.groovy (the test). Add the code in Listing 7 to DateTagLib.groovy:

Listing 7. A simple custom Grails tag

```
class DateTagLib {
  def thisYear = {
    out << Calendar.getInstance().get(Calendar.YEAR)
  }
}
```

Listing 7 creates a `<g:thisYear>` tag. As you can see, the year is written directly to the output stream. Listing 8 shows the new tag in action:

Listing 8. Copyright notice with a custom tag

```
<div id="copyright">
&copy; 2002 - <g:thisYear />, FakeCo Inc. All Rights Reserved.
</div>
```

You might think you're done at this point. I humbly suggest that you're only halfway there.

Testing TagLibs

Even though everything looks good for now, you should write a test to ensure that this tag doesn't break in the future. Michael Feathers, author of *Working Effectively with Legacy Code*, says that any code without a test is legacy code. To make sure that Mr. Feathers doesn't yell at you, add the code in Listing 9 to DateTagLibTests.groovy:

Listing 9. Test for a custom tag

```
class DateTagLibTests extends GroovyTestCase {
  def dateTagLib

  void setUp(){
    dateTagLib = new DateTagLib()
```

```
  }

  void testThisYear() {
    String expected = Calendar.getInstance().get(Calendar.YEAR)
    assertEquals("the years don't match", expected, dateTagLib.thisYear())
  }
}
```

A `GroovyTestCase` is a thin Groovy facade over a JUnit 3.x `TestCase`. Writing a test for a simple one-line tag might seem excessive, but you'd be surprised how many times the one-liners end up being the source of trouble. Writing the test isn't difficult, and it's better to be safe than sorry. Type `grails test-app` to run the test. If everything is okay, you should see the message shown in Listing 10:

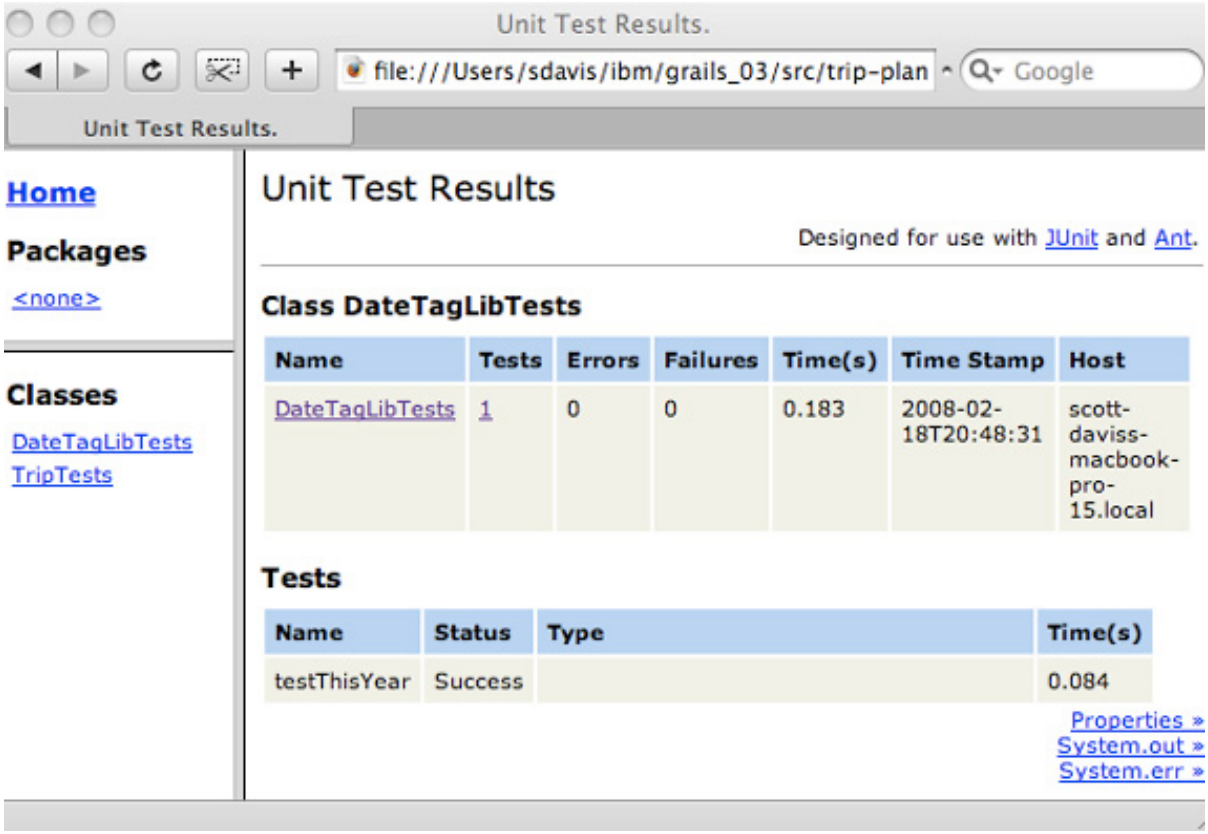
Listing 10. Passing tests in Grails

```
-------------------------------------------------------
Running 2 Integration Tests...
Running test DateTagLibTests...
                   testThisYear...SUCCESS
Running test TripTests...
                   testSomething...SUCCESS
Integration Tests Completed in 506ms
-------------------------------------------------------
```

If the appearance of `TripTests` takes you by surprise, don't worry. When you typed `grails create-domain-class Trip`, a test was generated on your behalf. As a matter of fact, every Grails `create` command generates a corresponding test. Yes, testing is *that* important in modern software development. If you aren't already in the habit of writing tests, allow Grails to nudge you gently in the right direction. You won't regret it.

The `grails test-app` command, in addition to running the tests, creates a nice HTML report. Open test/reports/html/index.html in a browser to see the standard JUnit test report, as shown in Figure 1.

Figure 1. Unit test report



Your simple custom tag is coded and tested. Now you'll build a tag that's a wee bit more sophisticated.

Advanced custom tags

More-sophisticated tags can work with attributes and the tag body. For example, the copyright solution as it stands now still requires too much copy/paste for my taste. I'd like to wrap up all of the current behavior in a truly reusable tag like this one: `<g:copyright startYear="2002">FakeCo Inc.</g:copyright>`. Listing 11 shows the code:

Listing 11. A Grails tag that works with attributes and the body

```
class DateTagLib {
  def thisYear = {
    out << Calendar.getInstance().get(Calendar.YEAR)
  }

  def copyright = { attrs, body ->
    out << "<div id='copyright'>"
    out << "&copy; ${attrs['startYear']} - ${thisYear()}, ${body()}"
    out << " All Rights Reserved."
    out << "</div>"
  }
}
```

Notice that `attrs` is a `HashMap` of the tag attributes. I use it here to grab the `startYear` attribute. I call the `thisYear` tag as a closure. (This is the same closure call that I could make from the GSP page in curly braces if I felt so inclined.) Similarly, the `body` is passed into the tag as a closure, so I call it the same way I'd call any other tag. This ensures that my custom tags can be nested to any arbitrary depth in the GSP.

You've probably noticed that custom TagLibs use the same `g:` namespace as the standard Grails TagLibs. If you'd like to place your TagLibs in a custom namespace, add `static namespace = 'trip'` to DateTagLib.groovy. In your GSP, the TagLib should now read `<trip:copyright startYear="2002">FakeCo Inc.</trip:copyright>`.

Partial templates

Custom tags are a great way to reuse short snippets of code that might otherwise end up in a copy/pasted scriptlet. For larger blocks of GSP markup, you can use a *partial template*.

Partial templates are officially called *templates* in the Grails documentation. The only problem is that the word *template* is overloaded to mean a couple of different things in Grails. As you'll see in the next section, you'll install the default templates to change the scaffolded views. Changes to these templates can include the partial templates that I'm about to talk about in this section. To help lessen the confusion, I borrow a bit of nomenclature from the Rails community and call these things *partial templates*, or often just *partials*.

A partial template is a chunk of GSP code that can be shared across multiple Web pages. For example, suppose I want a standard footer across all of my pages. To accomplish this, I'll create a partial named _footer.gsp. The leading underscore is a hint to the framework (as well as a visual cue for the developer) that this isn't a complete, well-formed GSP. If I create this file in the grails-app/views/trip directory, it's visible only to the `Trip` views. I'll store it in the grails-app/views directory so that it can be shared globally by all pages. Listing 12 shows the partial template for the globally shared footer:

Listing 12. A Grails partial template

```
<div id="footer">
  <g:copyright startYear='2002'>FakeCo, Inc.</g:copyright>

  <div id="powered-by">
    <img src="${createLinkTo(dir:'images', file:'grails-powered.jpg')}" />
  </div>
</div>
```

As you can see, a partial template allows you to express yourself in HTML/GSP syntax. In contrast, a custom TagLib is written in Groovy. Another way to keep the two straight in your mind is that TagLibs are generally better for encapsulating microbehavior, whereas partial templates are better for reusing layout elements.

For this example to work as written, you need to download the "Powered by Grails" button to the grails-app/web-app/images directory (see Resources). You'll see plenty of branding collateral on the download page, including everything from high-resolution logos to 16x16 favicons.

Listing 13 shows how you include your newly created footer in the bottom of your list.gsp page:

Listing 13. Rendering a partial template

```
<html><body>
...
<g:render template="/footer" />
</body></html>
```

Notice that you leave the underscore off when rendering a template. If you'd saved _footer.gsp in the trip directory, you'd leave the leading slash off as well. Think of it this way: the grails-app/views directory is the root of the view hierarchy.

Customizing the default scaffolding

Now that you have some good, testable, reusable components in place, you can make them a part of the default scaffolding. Recall that this is what gets dynamically generated when you put `def scaffold = Foo` in your controller. The default scaffolding also serves as the source for the GSPs that get created when you type `grails generate-views Trip` or `grails generate-all Trip`.

To customize the default scaffolding, type `grails install-templates`. This adds a new src/templates directory to the project. You should see three directories named artifacts, scaffolding, and war.

The artifacts directory holds the templates for various Groovy classes: `Controller`, `DomainClass`, `TagLib`, to name just a few. If, for example, you'd like all of your controllers to extend an abstract parent class, you can make the change here. All new controllers will be based on your modified template code. (Some people add `def scaffold = @artifact.name@` so that dynamic scaffolding is the default behavior for all of their controllers.)

The war directory contains the web.xml file familiar to all Java EE developers. If you

need to add your own parameters, filters, or servlets, this is the place to do it. (JSF enthusiasts: are you paying attention?) When you type `grails war`, the web.xml file found here is what gets included in the resulting WAR.

The scaffolding directory contains the raw material for the dynamically generated views. Open up list.gsp and add `<g:render template="/footer" />` to the bottom of the file. Because these templates are shared across all views, be sure to use global partial templates.

Now that you've tweaked the List view, it's time to verify that your changes are in effect. Modifications to the default templates are one of the few places that require you to restart the server. Once Grails is up and running again, visit http://localhost:9090/trip-planner/airline/list in a browser. If you are using default scaffolding in `AirlineController`, your new footer should appear at the bottom of the page.

Conclusion

That wraps up another installment of *Mastering Grails* . You should now know a bit more about GSP and the alternate view technologies available to Grails. You should have a better understanding of the default tags used in many of the generated pages. You should definitely feel a little bit dirty the next time you write a scriptlet, given how easy it is to do the right thing by writing a custom TagLib instead. You've seen how to create partial templates, and you've seen how easy it is to add them to the default scaffolded views.

Next month, your tour of the Grails Web framework will focus on Ajax. The ability to make "micro" HTTP requests without reloading the entire page is the secret sauce behind Google Maps, Flickr, and many other popular Web sites. You'll put a little of that same magic to use in Grails. Specifically, you'll create a many-to-many relationship and use Ajax to make the user experience natural and enjoyable.

Until then, have fun mastering Grails.

Downloads

Description	Name	Size	Download method
Sample code	j-grails03118.zip	256KB	HTTP

Information about download methods

Resources

Learn

- *Mastering Grails* : Read more in this series to gain a further understanding of Grails and all you can do with it.
- Grails: Visit the Grails Web site.
- Grails Framework Reference Documentation: The Grails bible.
- *Groovy Recipes* (Pragmatic Bookshelf, March 2008): Learn more about Groovy and Grails from the author's latest book.
- *Practically Groovy* : This developerWorks series is dedicated to exploring the practical uses of Groovy and teaching you when and how to apply them successfully.
- Groovy: Learn more about Groovy at the project Web site.
- AboutGroovy.com: Keep up with the latest Groovy news and article links.
- DRY: A process philosophy emphasizing that information should not be duplicated.
- Model-View-Controller: Grails follows this popular architectural pattern.
- Grails Plugins: Documentation for all Grails plug-ins.
- GSP Tag Reference: Your guide to the standard GSP tags.
- Dynamic Tag Libraries: Delve into custom tag writing.
- Testing Tag Libraries: Get a handle on integration testing for TagLibs.
- "Getting started with JavaServer Faces 1.2, Part 1: Building basic applications" (Richard Hightower, developerWorks, December 2007): Work through this example-rich tutorial series to get up to speed with JSF.
- "Ajax — A guide for the perplexed, Part 2: Develop a Dojo-based blog reader" (Gal Shachor, Ksenya Kveler, and Maya Barnea, developerWorks, December 2007): Put the Dojo toolkit through its paces with the help of the Atom protocol.
- "Ajax for Java developers: Exploring the Google Web Toolkit" (Philip McCarthy, developerWorks, June 2006): Learn about GWT's capabilities and work through a sample application.
- "Technology options for Rich Internet Applications" (Vaibhav V. Gadge, developerWorks, July 2006): Read an RIA overview.
- Technology bookstore: Browse 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.
- "Powered by Grails" button: You must use this button in this article's example application.

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.