Mastering Grails: The Grails event model

Customize behavior throughout the application life cycle

Skill Level: Introductory

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

12 Aug 2008

Everything in Grails, from build scripts to individual artifacts such as domain classes and controllers, throw events at key points during an application's life cycle. In this *Mastering Grails* installment, you'll learn how to set up listeners to catch these events and react to them with custom behavior.

Building a Web site is a study in event-driven, reactive development. Your application sits idle, anxiously waiting for the user to send in a request. It passes back the response and then goes back to sleep until the next call. In addition to the traditional Web life cycle of HTTP requests and responses, Grails provides a number of custom touch points where you can tap into the event model and provide behavior of your own.

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.

In this article, you'll discover the myriad of events being thrown in during the build process. You'll customize the startup and shutdown of your application. And finally, you'll explore the life cycle events of Grails domain classes.

Build events

Your first step as a Grails developer is to type `grails create-app`. The last thing you type is either `grails run-app` or `grails war`. These commands and everything you type in between all have events that get thrown at key points of the process.

Take a look at the $GRAILS_HOME/scripts directory. The files in this directory are Gant scripts that correspond to the commands you type in. For example, when you type `grails clean`, Clean.groovy is invoked.

The grooviness of Gant
You took your first glance at a Gant script in the first article of this series. Recall that Gant is thin Groovy facade over Apache Ant. Gant doesn't reimplement the Ant tasks — it literally calls the underlying Ant code to ensure maximum compatibility. Everything that you can do in Ant, you can do in Gant as well. The only difference is that Gant scripts are Groovy scripts instead of XML files. (See Resources for more information on Gant.)

Open Clean.groovy in a text editor. The first target you'll see is the `default` target, shown in Listing 1:

Listing 1. The default target in Clean.groovy

```
target ('default': "Cleans a Grails
project") {
   clean()
   cleanTestReports()
}
```

As you can see, there's not much to it. It runs the `clean` target and then the `cleanTestReports` target. Following the call stack, take a look at the `clean` target, shown in Listing 2:

Listing 2. The clean target in Clean.groovy

```
target ( clean: "Implementation of clean") {
    event("CleanStart", [])
    depends(cleanCompiledSources, cleanGrailsApp, cleanWarFile)
    event("CleanEnd", [])
}
```

If you want to customize the `clean` command's behavior, you can add your own code right here. However, the problem with that approach is that you must migrate your customizations each time you upgrade Grails. It also makes your build more fragile as you move from computer to computer. (The Grails installation files are

rarely checked into version control — just the application code.) To avoid the dreaded "but it works on *my* box" syndrome, I prefer to keep these types of customizations in the project. This ensures that every fresh checkout from source control contains all of the customizations needed for a successful build. It also helps keep things consistent if you are using a Continuous Integration server such as CruiseControl.

Notice that a couple of events are thrown during the `clean` target. `CleanStart` happens before the process starts, and `CleanEnd` happens afterwards. You can tap into these events in your project, keeping your custom code with the project code and leaving the Grails installation files untouched. All you need to do is create a listener.

Create a file named Events.groovy in the scripts directory of your project. Add the code shown in Listing 3:

Listing 3. Adding event listeners to Events.groovy

```
eventCleanStart = {
  println "### About to clean"
}

eventCleanEnd = {
  println "### Cleaning complete"
}
```

If you type `grails clean`, you should see output similar to Listing 4:

Listing 4. Console output showing your new comments

```
$ grails clean

Welcome to Grails 1.0.3 - http://grails.org/
Licensed under Apache Standard License 2.0
Grails home is set to: /opt/grails

Base Directory: /src/trip-planner2
Note: No plugin scripts found
Running script /opt/grails/scripts/Clean.groovy
Environment set to development
Found application events script
### About to clean
  [delete] Deleting: /Users/sdavis/.grails/1.0.3/projects/trip-planner2/resources/web.xml
  [delete] Deleting directory /Users/sdavis/.grails/1.0.3/projects/trip-planner2/classes
  [delete] Deleting directory /Users/sdavis/.grails/1.0.3/projects/trip-planner2/resources
### Cleaning complete
```

Of course, rather than writing simple messages to the console, you can do real work as well. Perhaps some additional directories should be deleted. Maybe you'd like to "reset" XML files by copying new ones over the existing ones. Anything you can do in Groovy (or Java programming), you can do here.

The CreateFile event

Here's another example of events you can tap into during the build process. Every time you type one of the `create-` commands (`create-controller`, `create-domain-class`, and so on), a `CreatedFile` event is fired. Take a look at scripts/CreateDomainClass.groovy, shown in Listing 5:

Listing 5. CreateDomainClass.groovy

```
Ant.property(environment:"env")
grailsHome = Ant.antProject.properties."env.GRAILS_HOME"

includeTargets << new File ( "${grailsHome}/scripts/Init.groovy" )
includeTargets << new File( "${grailsHome}/scripts/CreateIntegrationTest.groovy")

target ('default': "Creates a new domain class") {
    depends(checkVersion)

   typeName = ""
   artifactName = "DomainClass"
   artifactPath = "grails-app/domain"
   createArtifact()
   createTestSuite()
}
```

You can't see the `CreatedFile` event being called here, but look in the `createArtifact` target in $GRAILS_HOME/scripts/Init.groovy. (The `createTestSuite` target in $GRAILS_HOME/scripts/CreateIntegrationTest.groovy ends up calling the `createArtifact` target in $GRAILS_HOME/scripts/Init.groovy as well.) On the next-to-last line of the `createArtifact` target, you should see the following call: `event("CreatedFile", [artifactFile])`.

This event is different from the `CleanStart` event in one significant way: it passes a value back to the event handler. In this case, it is the full path to the file that just got created. (As you'll see in just a bit, the second argument is a list — you can pass back as many comma-delimited values as you'd like.) You must set up your event handler to catch the incoming value.

Suppose you would like to add these newly created files to your source control automatically. In Groovy, you can surround anything you'd normally type at the command line in quotes and call `execute()` on the `String`. Add the event handler in Listing 6 to scripts/Events.groovy:

Listing 6. Adding artifacts to Subversion automatically

```
eventCreatedFile = {fileName ->
  "svn add ${fileName}".execute()
  println "### ${fileName} was just added to Subversion."
}
```

Now type `grails create-domain-class Hotel` and look for the output. If you

aren't using Subversion, this command will fail silently. If you are using Subversion, type `svn status`. You should see the files added (both the domain class and the corresponding integration test).

Discovering the build events that are called

The quickest way to discover which events get thrown by what scripts is to search the Grails scripts for the `event()` call. On a UNIX® system, you can `grep` for the `event` string in the Groovy scripts, as shown in Listing 7:

Listing 7. Grepping for the event calls in Grails scripts

```
$ grep "event(" *.groovy
Bootstrap.groovy:       event("AppLoadStart", ["Loading Grails Application"])
Bootstrap.groovy:       event("AppLoadEnd", ["Loading Grails Application"])
Bootstrap.groovy:       event("ConfigureAppStart", [grailsApp, appCtx])
Bootstrap.groovy:       event("ConfigureAppEnd", [grailsApp, appCtx])
BugReport.groovy:    event("StatusFinal", ["Created bug-report ZIP at ${zipName}"])
```

Once you know what is being called, you can create a corresponding listener in scripts/Events.groovy and deeply customize your build environment.

Throwing custom events

Of course, now that you see how this works, there is nothing stopping you from adding your own events. If you absolutely need to customize the scripts in $GRAILS_HOME/scripts (as we're about to do to throw a custom event), I highly recommend copying them into the scripts directory in your project. This means that the custom script is checked into source control along with everything else. Grails asks you which version of the script you'd like to run — the one in $GRAILS_HOME or the one in the local scripts directory.

Copy $GRAILS_HOME/scripts/Clean.groovy to the local scripts directory and add the following event after the `CleanEnd` event:

```
event("TestEvent", [new Date(), "Some Custom Value"])
```

The first argument is the name of the event. The second argument is a list of items to return. In this case, you are returning a current datestamp and a custom message.

Add the closure in Listing 8 to scripts/Events.groovy:

Listing 8. Catching your custom event

```
eventTestEvent = {timestamp, msg ->
```

```
  println "### ${msg} occurred at ${timestamp}"
}
```

When you type `grails clean` and choose the local version of the script, you should see:

```
### Some Custom Value occurred at Wed Jul 09 08:27:04 MDT 2008
```

.

Bootstrap

In addition to build events, you can tap into application events. The grails-app/conf/BootStrap.groovy file runs every time Grails starts and stops. Open BootStrap.groovy in a text editor. The `init` closure gets invoked on startup. The `destroy` closure gets called when the application is shut down.

To begin, add some simple text to the closures, as shown in Listing 9:

Listing 9. Getting started with BootStrap.groovy

```
def init = {
  println "### Starting up"
}

def destroy = {
  println "### Shutting down"
}
```

Type `grails run-app` to start the application. You should see the `### Starting Up` message appear near the end of the process.

Now press **CTRL+C**. Did you see the `### Shutting Down` message? Me neither. The problem is that CTRL+C unceremoniously ends the server without calling the `destroy` closure. Rest assured that when your application server shuts down, this closure gets called. But you don't need to type `grails war` and load the WAR in Tomcat or IBM®WebSphere® to see the `destroy` event.

To see both the `init` and `destroy` events fire, start up Grails in interactive mode by typing `grails interactive`. Now type `run-app` to start the application and `exit` to shut down the server. Running in interactive mode speeds up the development process considerably because you keep the JVM running and warmed up. As a bonus, the application shuts down more gracefully than it does with the CTRL+C brute-force method.

Adding records to the database during bootstrap

What can you do with the BootStrap.groovy script besides simple console output? Commonly, folks use these hooks to insert records into the database.

To begin, add a name field to the `Hotel` class you created earlier, as shown in Listing 10:

Listing 10. Add a field to the Hotel class

```
class Hotel{
  String name
}
```

Now scaffold out a `HotelController`, as shown in Listing 11:

Listing 11. Create a Hotel Controller

```
class HotelController {
  def scaffold = Hotel
}
```

Note: If you disabled the `dbCreate` variable in grails-app/conf/DataSource.groovy as discussed in "Grails and legacy databases," you should add it back in and set it to `update` for this example. Your other option, of course, is to keep the `Hotel` table in sync with the changes to the `Hotel` class manually.

Now add the code in Listing 12 to BootStrap.groovy:

Listing 12. Saving and deleting records in BootStrap.groovy

```
def init = { servletContext ->
  new Hotel(name:"Marriott").save()
  new Hotel(name:"Sheraton").save()
}

def destroy = {
  Hotel.findByName("Marriott").delete()
  Hotel.findByName("Sheraton").delete()
}
```

For the next couple of examples, you might want to keep a MySQL console open and watch the database. Type `mysql --user=grails -p --database=trip` to log in. (Remember that the password is *server*.) Then perform the following steps:

1. Start up Grails if it isn't already running.
2. Type `show tables;` to confirm that the `Hotel` table was created.
3. Type `desc hotel;` to see the columns and data types.

4. Type select `* from hotel;` to confirm that the records were inserted.

5. Type `delete from hotel;` to delete all records.

Failsafe database inserts and deletes in BootStrap.groovy

You might want to be a bit more fail-safe when doing database inserts and deletes in BootStrap.groovy. If you don't check to see if the record exists before you insert it, you could end up with duplicates in the database. If you try to delete a record that doesn't exist, you'll begin seeing nasty exceptions being thrown at the console. Listing 13 shows how to perform failsafe inserts and deletes:

Listing 13. Failsafe inserts and deletes

```
def init = { servletContext ->
  def hotel = Hotel.findByName("Marriott")
  if(!hotel){
    new Hotel(name:"Marriott").save()
  }

  hotel = Hotel.findByName("Sheraton")
  if(!hotel){
    new Hotel(name:"Sheraton").save()
  }
}

def destroy = {
  def hotel = Hotel.findByName("Marriott")
  if(hotel){
    Hotel.findByName("Marriott").delete()
  }

  hotel = Hotel.findByName("Sheraton")
  if(hotel){
    Hotel.findByName("Sheraton").delete()
  }
}
```

If you call `Hotel.findByName("Marriott")` and that `Hotel` doesn't exist in the table, you'll get a `null` object in return. The next line, `if(!hotel)`, evaluates to `true` only for nonnull values. This ensures that you save the new `Hotel` only if it doesn't already exist. In the `destroy` closure, you perform the same tests to make sure that you aren't trying to delete records that don't exist.

Performing environment-specific actions in BootStrap.groovy

If you want something to happen only when you are running in a specific mode, the `GrailsUtil` class can help. Import `grails.util.GrailsUtil` at the top of the file. The static `GrailsUtil.getEnvironment()` method (shortened to simply `GrailsUtil.environment` thanks to Groovy's shorthand getter syntax) allows you to figure out what mode you are running in. Couple this knowledge with a `switch` statement, as shown in Listing 14, and you can make environment-specific

behavior happen when Grails starts up:

> **Groovy's robust switch**
> Notice that Groovy's `switch` statement is more robust than the Java `switch` statement. In Java code, you can switch only on integer values. In Groovy, you can switch on `String` values as well.

Listing 14. Environment-specific actions in BootStrap.groovy

```
import grails.util.GrailsUtil

class BootStrap {

     def init = { servletContext ->
       switch(GrailsUtil.environment){
         case "development":
           println "#### Development Mode (Start Up)"
           break
         case "test":
           println "#### Test Mode (Start Up)"
           break
         case "production":
           println "#### Production Mode (Start Up)"
           break
       }
     }

     def destroy = {
       switch(GrailsUtil.environment){
         case "development":
           println "#### Development Mode (Shut Down)"
           break
         case "test":
           println "#### Test Mode (Shut Down)"
           break
         case "production":
           println "#### Production Mode (Shut Down)"
           break
       }
     }
}
```

You now have what it takes to insert records only in test mode. But don't stop there. I often externalize my test data in XML files. Combine what you learned here with the XML backup and restore scripts from "Grails and legacy databases," and you have a powerful testbed in place.

Because BootStrap.groovy is an executable script instead of a passive configuration file, you can do literally anything you can do in Groovy. Perhaps you want to make a Web services call on startup, announcing to a central server that this instance is up and running. Maybe you want to synchronize local lookup tables from a common source. The possibilities are truly limitless.

Micro-level events

Now that you've seen some macro-level events, here are some micro-level events.

Timestamping your domain classes

If you provide a couple of specially named fields, GORM automatically timestamps your class, as shown in Listing 15:

Listing 15. Timestamp fields

```
class Hotel{
  String name
  Date dateCreated
  Date lastUpdated
}
```

As the names imply, the `dateCreated` field is populated the first time the data is inserted into the database. The `lastUpdated` field is populated each subsequent time the database record is updated.

To verify that these fields are getting populated behind the scenes, do one more thing: disable them in create and edit views. You could do this by typing `grails generate-views Hotel` and deleting the fields in the create.gsp and edit.gsp files, but here's a way to be a bit more dynamic with your scaffolded views. In "Changing the view with Groovy Server Pages," I had you type `grails install-templates` so that you could tweak the scaffolded views. Look in scripts/templates/scaffolding for create.gsp and edit.gsp. Now add the two timestamp fields to the `excludedProps` list in the templates, as shown in Listing 16:

Listing 16. Excluding timestamp fields from default scaffolding

```
excludedProps = ['dateCreated','lastUpdated',
                 'version',
                 'id',
                   Events.ONLOAD_EVENT,
                   Events.BEFORE_DELETE_EVENT,
                   Events.BEFORE_INSERT_EVENT,
                   Events.BEFORE_UPDATE_EVENT]
```

This suppresses the creation of fields in the create and edit views but still leaves the fields in the list and show views. Create a `Hotel` or two and verify that the fields are getting updated automatically.

If your application is already using those field names, you can easily disable this feature, as shown in Listing 17:

Listing 17. Disabling timestamps

```
static mapping = {
  autoTimestamp false
}
```

Recall from "Grails and legacy databases" that you can also specify `version false` if you want to disable the automatic creation and updating of the `version` field, used for optimistic locking.

Adding event handlers to domain classes

In addition to timestamping your domain class, you can tap into four event hooks: `beforeInsert`, `befortUpdate`, `beforeDelete`, and `onLoad`.

The names of these closures should speak for themselves. The `beforeInsert` closure gets called before the `save()` method. The `beforeUpdate` closure gets called before the `update()` method. The `beforeDelete` closure gets called before the `delete()` method. Finally, `onLoad` gets called when the class gets loaded from the database.

Let's say that your company already has a policy for timestamping database records and has standardized the names of these fields to `cr_time` and `up_time`. You have a couple of options to make Grails comply with this corporate policy. One is to use the static-mapping trick you learned in "Grails and legacy databases" to wire the default Grails field names to the default company column names, as shown in Listing 18:

Listing 18. Mapping the timestamp fields

```
class Hotel{
  Date dateCreated
  Date lastUpdated

  static mapping = {
    columns {
      dateCreated column: "cr_time"
      lastUpdated column: "up_time"
    }
  }
}
```

The other option is to name the fields in the domain class to match the corporate column names and create `beforeInsert` and `beforeUpdate` closures to populate the fields, as shown in Listing 19. (Don't forget to make the new fields `nullable` — otherwise the `save()` method will fail silently in BootStrap.groovy.)

Listing 19. Adding beforeInsert and beforeUpdate closures

```
class Hotel{
  static constraints = {
    name()
    crTime(nullable:true)
    upTime(nullable:true)
  }

  String name
  Date crTime
  Date upTime

  def beforeInsert = {
    crTime = new Date()
  }

  def beforeUpdate = {
    upTime = new Date()
  }
}
```

Start and stop your application a few more times, making sure that the new fields get populated as expected.

Like all of the other events you've seen up to this point, what you choose to do with them is completely up to you. Recall from "Grails services and Google Maps" that you created a `Geocoding` service to convert street addresses to latitude/longitude points so that an `Airport` could be plotted on a map. In that article, I had you call the service in the `save` and `update` closures in the `AirportController`. I'd be tempted to move that service call to `beforeInsert` and `beforeUpdate` in the `Airport` class to make it happen transparently and automatically.

How can you share this behavior across all of your classes? I'd add these fields and closures to the default `DomainClass` template in src/templates. That way every new domain class created already has the fields and event closures in place.

Conclusion

Events throughout Grails can help you further customize the way your application behaves. You can extend the build process without modifying the standard Grails scripts by creating an Events.groovy file in the scripts directory. You can customize the startup and shutdown process by adding your own code to the `init` and `destroy` closures in the BootStrap.groovy file. And finally, adding closures like `beforeInsert` and `beforeUpdate` to your domain class allows you to add behavior such as timestamping and geocoding.

In the next article, I'll introduce you to the idea of using Grails to create Web services based on Representational State Transfer (REST) of your data. You'll see how Grails easily supports the HTTP `GET`, `PUT`, `POST`, and `DELETE` actions needed to support next-generation, RESTful Web services. Until then, keep mastering Grails.

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: Learn more about Groovy and Grails in Scott Davis' 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.
- "Build software with Gant" (Andrew Glover, developerWorks, May 2008): Read this tutorial to find out how the marriage of Groovy and Apache Ant makes flexible builds easier.
- 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.

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

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