Mastering Grails: Grails and legacy databases

Can convention over configuration survive unconventional databases schemas?

Skill Level: Introductory

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

15 Jul 2008

In this *Mastering Grails* installment, Scott Davis explores the various ways that Grails can use database tables that don't conform to the Grails naming standard. If you have Java™ classes that already map to your legacy databases, Grails allows you to use them unchanged. You'll see examples that use Hibernate HBM files and Enterprise JavaBeans 3 annotations with legacy Java classes.

The Grails Object Relational Mapping (GORM) API is one of the centerpieces of the Grails Web framework. In "GORM: Funny name, serious technology," you were introduced to the basics of GORM, including simple one-to-many relationships. Later, in "Many-to-many relationships with a dollop of Ajax," you used GORM to model increasingly sophisticated class relationships. Now you'll see how the "ORM" in GORM has the flexibility to deal with table and column names in your legacy databases that don't follow standard GORM naming conventions.

Backing up and restoring your data

Any time you are dealing with existing data in a database, it's crucial to have an up-to-date backup. Murphy of Murphy's Law fame seems to be my patron saint. Anything that can go wrong will, so it's best to be prepared for the worst.

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.

Backing up

In addition to backing up the target database with your normal backup software, I suggest keeping a second copy of the data in plain text. This allows you to create test and development databases easily with the same dataset, as well as move the data across database servers (for example, from MySQL to DB2 and back again) with ease.

Once again, you'll work with the Trip Planner application you've been developing throughout this series. Listing 1 is a Groovy script named backupAirports.groovy that backs up the records from the `airport` table. In three statements and fewer than 20 lines of code, it connects to the database, selects every row from the table, and exports the data as XML.

Listing 1. backupAirports.groovy

```
sql = groovy.sql.Sql.newInstance(
   "jdbc:mysql://localhost/trip?autoReconnect=true",
   "grails",
   "server",
   "com.mysql.jdbc.Driver")

x = new groovy.xml.MarkupBuilder()

x.airports{
  sql.eachRow("select * from airport order by id"){ row ->
    airport(id:row.id){
      version(row.version)
      name(row.name)
      city(row.city)
      state(row.state)
      country(row.country)
      iata(row.iata)
      lat(row.lat)
      lng(row.lng)
    }
  }
}
```

The first statement in Listing 1 creates a new `groovy.sql.Sql` object. This is a thin Groovy facade over the standard flock of JDBC classes, including `Connection`, `Statement`, and `ResultSet`. You probably recognize the four arguments to the `newInstance` factory method: JDBC connection string, username, password, and JDBC driver. (These are the same values found in grails-app/conf/DataSource.groovy.)

The next statement creates a `groovy.xml.MarkupBuilder`. This class allows

you to create XML documents on the fly.

The final statement (beginning with `x.airports`) creates the XML tree. The root element in the XML document is `airports`. For each row in the database, an `airport` element will be created with an `id` attribute. Nested within the `airport` element are `version`, `name`, and `city` elements. (For more information on Groovy `Sql` and `MarkupBuilder` usage, see Resources).

Listing 2 shows the resulting XML:

Listing 2. XML output from backup script

```
<airports>
  <airport id='1'>
    <version>2</version>
    <name>Denver International Airport</name>
    <city>Denver</city>
    <state>CO</state>
    <country>US</country>
    <iata>den</iata>
    <lat>39.8583188</lat>
    <lng>-104.6674674</lng>
  </airport>
  <airport id='2'>...</airport>
  <airport id='3'>...</airport>
</airports>
```

In the backup script, it's important that you pull the records out in primary-key order. When you restore this data, the values must be inserted in the same order to ensure that foreign-key values still match up. (I'll talk about this in further detail in the next section.)

Note that the script is completely independent of the Grails framework. For it to work, you must have Groovy installed on your system. (See Resources for download and installation instructions.) You must also have the JDBC driver JAR on your classpath. You can specify it as you run the script. On UNIX®, type:

```
groovy -classpath /path/to/mysql.jar:. backupAirports.groovy
```

Of course, file paths and JAR separators are different on Windows®. On Windows, type this instead:

```
groovy -classpath c:\path\to\mysql.jar;. backupAirports.groovy
```

I use MySQL frequently enough that I keep a copy of the JAR in the .groovy/lib directory in my home directory (/Users/sdavis on UNIX, c:\Documents and Settings\sdavis on Windows). JARs in this directory are automatically included in the classpath when you run Groovy scripts from the command line.

The script in Listing 1 writes the output to the screen. To save the data to a file, redirect the output when you run the script:

```
groovy backupAirports.groovy > airports.xml
```

Restoring the data

Getting the data out of the database is only half the battle. Getting it back in is just as important. The restoreAirports.groovy script, shown in Listing 3, reads in the XML using a Groovy `XmlParser`, constructs a SQL `insert` statement, and uses a Groovy SQL object to execute the statement. (For more information on `XmlParser`, see Resources.)

Listing 3. Groovy script to restore database records from XML

```
if(args.size()){
   f = new File(args[0])
   println f

   sql = groovy.sql.Sql.newInstance(
      "jdbc:mysql://localhost/aboutgroovy?autoReconnect=true",
      "grails",
      "server",
      "com.mysql.jdbc.Driver")

   items = new groovy.util.XmlParser().parse(f)
   items.item.each{item ->
     println "${item.@id} -- ${item.title.text()}"
      sql.execute(
         "insert into item (version, title, short_description, description,
                 url, type, date_posted, posted_by) values(?,?,?,?,?,?,?,?)",
         [0, item.title.text(), item.shortDescription.text(), item.description.text(),
             item.url.text(), item.type.text(), item.datePosted.text(),
             item.postedBy.text()]
         )
   }
}
else{
   println "USAGE: itemsRestore [filename]"
}
```

To run this script, type:

```
groovy restoreAirports.groovy airports.xml
```

Remember that for relationships between tables to work, the primary-key field on the *one* side of the relationship must match up to the foreign-key field on the *many* side of the relationship. For example, the value stored in the `id` column in the `airport` table must be the same as the value in the `arrival_airline_id` column in the `flight` table.

Transforming the USGS data

The USGS provides the airport data as a *shapefile*. This is a

well-established file format for exchanging geographic data. Three files, at minimum, make up a shapefile. The .shp file contains the geographic data — in this case, the latitude/longitude points for each airport. The .shx file is a spatial index. The .dbf file is — you guessed it — a good old dBase file that contains all of the nonspatial data (in this case, the airport name, the IATA code, and so on).

I used the Geospatial Data Abstraction Library (GDAL) — an open source set of command-line tools for manipulating geographic data — to convert the shapefile into a Geography Markup Language (GML) file used in this article. The specific command I used was:

```
ogr2ogr -f "GML" airports.xml airprtx020.shp
```

You can also convert the data to comma-separated value (CSV), GeoJSON, Keyhole Markup Language (KML), and a variety of other formats using GDAL.

To ensure that the autonumbered `id` fields get restored to the same value, be sure to drop all tables before you restore them. This resets the autonumbering back to 0 when Grails recreates the tables the next time it starts up.

Now that you have your airport data safely backed up (and presumably the data from the other tables as well), you are ready to begin experimenting with some new "legacy" data. Confused? The next section should help clarify things.

Importing new airport data

The United States Geological Survey (USGS) publishes a comprehensive list of airports in the United States, including the IATA codes and latitude/longitude points (see Resources). Not surprisingly, the USGS fields don't match up with the `Airport` class that is already in use. Although you could change the Grails class to match the names in the USGS table, that could involve a pretty extensive rewrite of the application. Instead, you'll explore a couple of different techniques that will allow your existing `Airport` class to map seamlessly to the new, different table schema behind the scenes.

To start, you need to get the USGS "legacy" data imported into your database. Run the createUsgsAirports.groovy script in Listing 4 to create the new table. (This script assumes that you are using MySQL. Because each database has slightly different syntax for creating tables, you might need to tweak it for other databases.)

Listing 4. Creating the USGS Airports table

```
sql = groovy.sql.Sql.newInstance(
   "jdbc:mysql://localhost/trip?autoReconnect=true",
   "grails",
   "server",
```

```
   "com.mysql.jdbc.Driver")

ddl = """
CREATE TABLE usgs_airports (
  airport_id bigint(20) not null,
  locid varchar(4),
  feature varchar(80),
  airport_name varchar(80),
  state varchar(2),
  county varchar(50),
  latitude varchar(30),
  longitude varchar(30),
  primary key(airport_id)
);
"""

sql.execute(ddl)
```

Take a look at usgs-airports.xml in Listing 5. It is an example of the GML format. This XML is a bit more complex than the simple XML in Listing 2 created by the backup script. Every element is in a namespace, and the elements are more deeply nested.

Listing 5. USGS airports in GML

```
<?xml version="1.0" encoding="utf-8" ?>
<ogr:FeatureCollection
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://ogr.maptools.org/airports.xsd"
     xmlns:ogr="http://ogr.maptools.org/"
     xmlns:gml="http://www.opengis.net/gml">

  <gml:featureMember>
    <ogr:airprtx020 fid="F0">
      <ogr:geometryProperty>
        <gml:Point>
          <gml:coordinates>-156.042831420898438,19.73573112487793</gml:coordinates>
        </gml:Point>
      </ogr:geometryProperty>
      <ogr:AREA>0.000</ogr:AREA>
      <ogr:PERIMETER>0.000</ogr:PERIMETER>
      <ogr:AIRPRTX020>1</ogr:AIRPRTX020>
      <ogr:LOCID>KOA</ogr:LOCID>
      <ogr:FEATURE>Airport</ogr:FEATURE>
      <ogr:NAME>Kona International At Keahole</ogr:NAME>
      <ogr:TOT_ENP>1271744</ogr:TOT_ENP>
      <ogr:STATE>HI</ogr:STATE>
      <ogr:COUNTY>Hawaii County</ogr:COUNTY>
      <ogr:FIPS>15001</ogr:FIPS>
      <ogr:STATE_FIPS>15</ogr:STATE_FIPS>
    </ogr:airprtx020>
  </gml:featureMember>

  <gml:featureMember>...</gml:featureMember>
  <gml:featureMember>...</gml:featureMember>
</ogr:FeatureCollection>
```

Now create the restoreUsgsAirports.groovy script, shown in Listing 6. To get at the namespaced elements, you need to declare a couple of `groovy.xml.Namespace` variables. Instead of using the simple dotted notation used in the previous

restoreAirport.groovy script (Listing 3), you must surround namespaced elements with square brackets.

Listing 6. Restoring the USGS Airport data to the database

```
if(args.size()){
  f = new File(args[0])
  println f

  sql = groovy.sql.Sql.newInstance(
     "jdbc:mysql://localhost/trip?autoReconnect=true",
     "grails",
     "server",
     "com.mysql.jdbc.Driver")

  FeatureCollection = new groovy.util.XmlParser().parse(f)
  ogr = new groovy.xml.Namespace("http://ogr.maptools.org/")
  gml = new groovy.xml.Namespace("http://www.opengis.net/gml")

  FeatureCollection[gml.featureMember][ogr.airprtx020].each{airprtx020 ->
   println "${airprtx020[ogr.LOCID].text()} -- ${airprtx020[ogr.NAME].text()}"
   points = airprtx020[ogr.geometryProperty][gml.Point][gml.coordinates].text().split(",")

     sql.execute(
        "insert into usgs_airports (airport_id, locid, feature, airport_name, state,
        county, latitude, longitude) values(?,?,?,?,?,?,?,?)",
        [airprtx020[ogr.AIRPRTX020].text(),
        airprtx020[ogr.LOCID].text(),
        airprtx020[ogr.FEATURE].text(),
        airprtx020[ogr.NAME].text(),
        airprtx020[ogr.STATE].text(),
        airprtx020[ogr.COUNTY].text(),
        points[1],
        points[0]]
        )
  }
}
else{
   println "USAGE: restoreAirports [filename]"
}
```

Type the following at the command prompt to insert the data in the usgs_airports.xml file into the newly created table:

```
groovy restoreUsgsAirports.groovy
usgs-airports.xml
```

To verify that this worked correctly, log in to MySQL from the command line and ensure the data is there, as shown in Listing 7:

Listing 7. Verifying the USGS airport data in the database

```
$ mysql --user=grails -p --database=trip

mysql> desc usgs_airports;
+--------------+-------------+------+-----+---------+-------+
| Field        | Type        | Null | Key | Default | Extra |
+--------------+-------------+------+-----+---------+-------+
```

```
| airport_id   | bigint(20)  | NO   | PRI |         |       |
| locid        | varchar(4)  | YES  |     | NULL    |       |
| feature      | varchar(80) | YES  |     | NULL    |       |
| airport_name | varchar(80) | YES  |     | NULL    |       |
| state        | varchar(2)  | YES  |     | NULL    |       |
| county       | varchar(50) | YES  |     | NULL    |       |
| latitude     | varchar(30) | YES  |     | NULL    |       |
| longitude    | varchar(30) | YES  |     | NULL    |       |
+--------------+-------------+------+-----+---------+-------+
8 rows in set (0.01 sec)

mysql> select count(*) from usgs_airports;
+----------+
| count(*) |
+----------+
|      901 |
+----------+
1 row in set (0.44 sec)

mysql> select * from usgs_airports limit 1\G
*************************** 1. row ***************************
  airport_id: 1
       locid: KOA
     feature: Airport
airport_name: Kona International At Keahole
       state: HI
      county: Hawaii County
    latitude: 19.73573112487793
   longitude: -156.042831420898438
```

Disabling dbCreate

Now that your legacy table is in place, you should do one last thing: disable the `dbCreate` variable in grails-app/conf/DataSource.groovy. Recall from "GORM: Funny name, serious technology" that this variable instructs GORM to create the corresponding table behind the scenes if it doesn't exist, and alter any existing tables to match the Grails domain classes. If you are dealing with legacy tables, you should disable this feature so that GORM won't disrupt the schema that other applications might be expecting.

It would be nice if you could selectively enable `dbCreate` for certain tables and disable it for others. Unfortunately, it is a global "all or nothing" setting. In situations in which I have a mix of new and legacy tables, I allow GORM to create the new tables, then turn `dbCreate` off and import my existing, legacy tables. You can see how having a good backup-and-restore strategy is important in such situations.

Static mapping blocks

The first strategy I'll demonstrate for mapping your domain class to a legacy table is using a static `mapping` block. This is what I use in most cases because it feels the most Grails-like. I'm used to adding static `constraints` blocks to my domain classes, so adding a static `mapping` block feels consistent with the rest of the framework.

Copy the grails-app/domain/Airport.groovy file to grails-app/domain/AirportMapping.groovy. This name is only for demonstration purposes. You're going to have three classes that all map back to the same table, so you need a way to keep each class uniquely named. (This is something that is not likely to happen in a real application.)

Comment out the city and country fields, because they don't exist in the new table. Remove them from the `constraints` block as well. Now add in the static `mapping` block to link the Grails names to the database names, as shown in Listing 8:

Listing 8. AirportMapping.groovy

```
class AirportMapping{
  static constraints = {
    name()
    iata(maxSize:3)
    state(maxSize:2)
    lat()
    lng()
  }

  static mapping = {
    table "usgs_airports"
    version false
    columns {
      id column: "airport_id"
      name column: "airport_name"
      iata column: "locid"
      state column: "state"
      lat column: "latitude"
      lng column: "longitude"
    }
  }

  String name
  String iata
  //String city
  String state
  //String country = "US"
  String lat
  String lng

  String toString(){
    "${iata} - ${name}"
  }
}
```

The first statement in the `mapping` block links the `AirportMapping` class to the `usgs_airports` table. The next statement tells Grails that the table does not have a `version` column. (GORM normally creates one to help facilitate optimistic locking.) Finally, the `columns` block maps the Grails names to the database names.

Notice that by using this mapping technique, you can ignore specific fields in the table. In this case, the `feature` and `county` columns aren't represented in the domain class. To do the opposite — have fields in a domain class that aren't stored in the table — add a static `transients` line. This line looks similar to the

`belongsTo` variable used in one-to-many relationships. For example, if there were two fields in the `Airport` class that didn't need to be stored in the table, the code would look like this:

```
static transients = ["tempField1", "tempField2"]
```

The `mapping` block demonstrated here just scratches the surface of what you can do with this technique. For more information, see Resources.

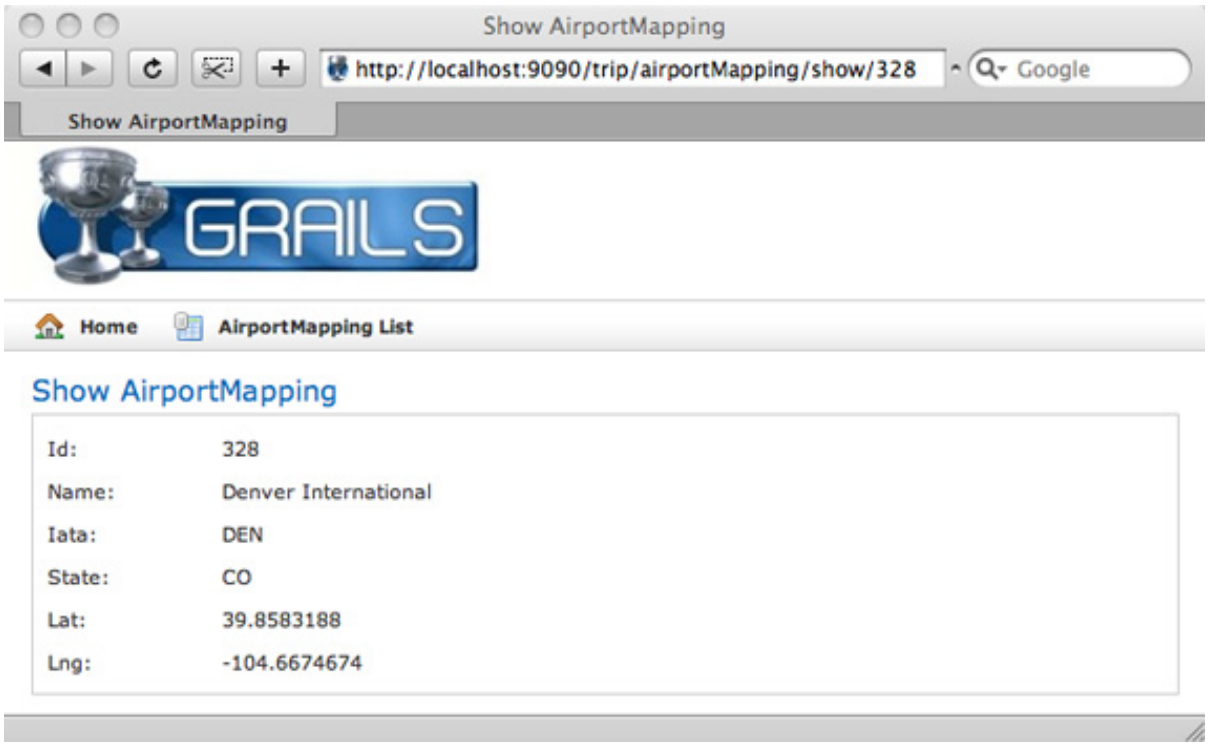
Making your legacy table read-only

Type `grails generate-all AirportMapping` to create the controller and GSP views. Because this table is essentially a look-up table, go into grails-app/controllers/AirportMappingController.groovy and leave only the `list` and `show` closures. Remove `delete`, `edit`, `update`, `create`, and `save`. (And don't forget to remove `delete`, `edit`, and `save` from the `allowedMethods` variable. You can remove the line entirely or leave an empty set of square brackets.)

A couple of quick changes are necessary to make the views read-only. First, remove the `New AirportMapping` link from the top of grails-app/views/airportMapping/list.gsp. Do the same for grails-app/views/airportMapping/show.gsp. Finally, remove the `edit` and `delete` buttons from the bottom of show.gsp.

Type `grails run-app` to verify that your `mapping` block works. You should see the page shown in Figure 1:

Figure 1. Verifying that the mapping block works



Using legacy Java classes with Hibernate mapping files

Now that you understand the `mapping` block, let's take things one step further. It's not hard to imagine that if you have legacy tables, you might have legacy Java classes as well. These next two mapping techniques assume that you'd like to mix existing Java code in with data from existing tables.

Before Java 1.5 brought annotations into the mix, Hibernate users created XML mapping files called HBM files. Recall that GORM is a thin Groovy facade over Hibernate, so it shouldn't surprise you that all of your old Hibernate tricks will still work.

To start, copy your legacy Java source files into src/java. If you are using packages, create one directory per package name. For example, the AirportHbm.java file shown in Listing 9 is in the `org.davisworld.trip` package. This means that the full path to the file should be src/java/org/davisworld/trip/AirportHbm.java.

Listing 9. AirportHbm.java

```
package org.davisworld.trip;

public class AirportHbm {
  private long id;
  private String name;
  private String iata;
  private String state;
```

```
  private String lat;
  private String lng;

  public long getId() {
    return id;
  }

  public void setId(long id) {
    this.id = id;
  }

  // all of the other getters/setters go here
}
```

Once your Java file is in place, you can create a "shadow" file next to it named AirportHbmConstraints.groovy, shown in Listing 10. This is where you can put the static `constraints` block that would normally be in the domain class. Be sure that it's in the same package as the Java class.

Listing 10. AirportHbmConstraints.groovy

```
package org.davisworld.trip

static constraints = {
  name()
  iata(maxSize:3)
  state(maxSize:2)
  lat()
  lng()
}
```

Files under the src directory will be compiled when you run the application or create the WAR file for deployment. If your Java code is already compiled, you can also simply JAR it up and place it in the lib directory.

Next, let's set up a controller. Following convention over configuration, the controller should be named AirportHbmController.groovy. Because the Java class is in a package, you can either place the controller in the same package or import the Java class at the top of the file. I prefer the import method, as shown in Listing 11:

Listing 11. AirportHbmController.groovy

```
import org.davisworld.trip.AirportHbm

class AirportHbmController {
  def scaffold = AirportHbm
}
```

Next, copy your existing HBM files into grails-app/conf/hibernate. You should have a single hibernate.cfg.xml file, shown in Listing 12, where you specify each mapping file used for each class. In this case, it should have an entry for the AirportHbm.hbm.xml file.

Listing 12. hibernate.cfg.xml

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
        <mapping resource="AirportHbm.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

Each class must have its own HBM file. This file is the XML equivalent of the static `mapping` block you used earlier. Listing 13 shows AirportHbm.hbm.xml:

Listing 13. AirportHbm.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="org.davisworld.trip.AirportHbm" table="usgs_airports">
        <id name="id" column="airport_id">
            <generator class="native"/>
         </id>
        <property name="name" type="java.lang.String">
            <column name="airport_name" not-null="true" />
        </property>
        <property name="iata" type="java.lang.String">
            <column name="locid" not-null="true" />
        </property>
        <property name="state" />
        <property name="lat" column="latitude" />
        <property name="lng" column="longitude" />
    </class>
</hibernate-mapping>
```

Notice that the full package name is specified in references to the Java class. The rest of the entries map the Java names to the table names. The `name` and `iata` field entries demonstrate the long form. Because the `state` field is the same in both the Java code and the table, you can shorten its entry; the last two fields — `lat` and `lng` — demonstrate the shortcut syntax. (For more on Hibernate mapping files, see Resources.)

Restart Grails if it is still running. You should now be able to see your Hibernate-mapped data at http://localhost:8080/trip/airportHbm.

Using Enterprise JavaBeans (EJB) 3 annotations on Java classes

As I mentioned earlier, Java 1.5 introduced annotations to the language. Annotations

allow you to add metadata directly to your Java class by prefixing it with an @. When Groovy 1.0 was released in December, 2006, it didn't support Java 1.5 language features such as annotations. That all changed when Groovy 1.5 was released a year later. This means that you can bring in your EJB3-annotated Java files to an existing Grails application as well.

Start once again with your EJB3-annotated Java file. Place AirportAnnotation.java, shown in Listing 14, in src/java/org.davisworld.trip, right next to the AirportHbm.java file:

Listing 14. AirportAnnotation.java

```
package org.davisworld.trip;

import javax.persistence.*;

@Entity
@Table(name="usgs_airports")
public class AirportAnnotation {
  private long id;
  private String name;
  private String iata;
  private String state;
  private String lat;
  private String lng;

  @Id
  @Column(name="airport_id", nullable=false)
  public long getId() {
    return id;
  }

  @Column(name="airport_name", nullable=false)
  public String getName() {
    return name;
  }

  @Column(name="locid", nullable=false)
  public String getIata() {
    return iata;
  }

  @Column(name="state", nullable=false)
  public String getState() {
    return state;
  }

  @Column(name="latitude", nullable=false)
  public String getLat() {
    return lat;
  }

  @Column(name="longitude", nullable=false)
  public String getLng() {
    return lng;
  }

  // The setter methods don't have an annotation on them.
  // They are not shown here, but they should be in the file
  //   if you want to be able to change the values.

}
```

Notice that you must import the `javax.persistence` package at the top of the file. `@Entity` and `@Table` annotate the class declaration, mapping it to the appropriate database table. The rest of the annotations appear above the getter method for each field. All fields should have a `@Column` annotation that maps the field name to the column name. The primary key should also have an `@ID` annotation.

The AirportAnnotationConstraints.groovy file in Listing 15 is no different from the previous example in Listing 10:

Listing 15. AirportAnnotationConstraints.groovy

```
package org.davisworld.trip

static constraints = {
  name()
  iata(maxSize:3)
  state(maxSize:2)
  lat()
  lng()
}
```

The AirportAnnotationController.groovy (shown in Listing 16) is scaffolded out in the usual manner:

Listing 16. AirportAnnotationController.groovy

```
import org.davisworld.trip.AirportAnnotation

class AirportAnnotationController {
  def scaffold = AirportAnnotation
}
```

The hibernate.cfg.xml file comes into play once again. This time, the syntax is slightly different. Instead of pointing it to an HBM file, you point it to the class directly, as shown in Listing 17:

Listing 17. hibernate.cfg.xml

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
        <mapping resource="AirportHbm.hbm.xml"/>
        <mapping class="org.davisworld.trip.AirportAnnotation"/>
    </session-factory>
</hibernate-configuration>
```

You need to do one final thing to get annotations to work. Grails isn't configured out of the box to look for EJB3 annotations. You must import a specific class in grails-app/conf/DataSource.groovy, as shown in Listing 18:

Listing 18. DataSource.groovy

```
import org.codehaus.groovy.grails.orm.hibernate.cfg.GrailsAnnotationConfiguration
dataSource {
  configClass = GrailsAnnotationConfiguration.class

   pooled = false
   driverClassName = "com.mysql.jdbc.Driver"
   username = "grails"
   password = "server"
}
```

Once you import `org.codehaus.groovy.grails.orm.hibernate.cfg.GrailsAnnotationConfiguratio` and allow Spring to inject it into the `dataSource` block as `configClass`, Grails will support EJB3 annotations as well as it supports HBM files and native mapping blocks.

If you forget this final step (I do nearly every time I use EJB3 annotations in Grails), you'll get the following error message:

Listing 19. Exception thrown if you don't inject configClass in DataSource.groovy

```
org.hibernate.MappingException:
An AnnotationConfiguration instance is required to use
<mapping class="org.davisworld.trip.AirportAnnotation"/>
```

Conclusion

At this point, *mapping* your *objects* to *relational* databases in *Grails* should be a breeze. (That's why they named it *GORM*, after all.) Once you feel confident that you can easily back up and restore your data, you have a variety of ways of getting Grails to conform to the nonstandard naming conventions in your legacy databases. The static `mapping` block is the easiest way to accomplish this task because it is the most Grails-like. But if you have legacy Java classes already mapped to your legacy database, there is no sense in reinventing the wheel. Whether you are using HBM files or the newer EJB3 annotations, Grails can take advantage of the work you've already done and allow you to move forward to other tasks.

In the next article, you'll get a chance to play around with the Grails event model. Everything from the build scripts to the individual Grails artifacts (domain classes, controllers, etc.) throw events at key points during the application's life cycle. You'll

learn how to set up listeners to catch these events and react with custom behavior. Until then, have fun Mastering Grails.

Downloads

Description	Name	Size	Download method
Sample code[1]	j-grails07158.zip	991KB	HTTP

Information about download methods

Note

1. The code download contains the Grails application, Groovy scripts, and the USGS Airport file in GML.

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 SQL": Get a handle on Groovy SQL from this tutorial.
- GroovyMarkup: Get started with GroovyMarkup.
- `XmlParser`: Read XML using Groovy's `XmlParser`.
- GORM - Mapping DSL: Read about mapping Grails domain classes to legacy schemas via a domain-specific language.
- Hibernate mapping files and EJB3 annotations with Hibernate: Read about these topics in the Hibernate documentation.
- *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.
- *Groovy Recipes*: Learn more about Groovy and Grails in Scott Davis' latest book.
- 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

- USGS Airports download: Download raw airport data from the USGS.
- Grails: Download the latest Grails release.
- Groovy: Download and install Groovy.

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.
UNIX is a registered trademark of The Open Group in the United States and other countries.
Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.