Mastering Grails: GORM: Funny name, serious technology

Understanding databases and Grails

Skill Level: Introductory

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

12 Feb 2008

Any good Web framework needs a solid persistence strategy. In this second installment of his *Mastering Grails* series, Scott Davis introduces the Grails Object Relational Mapping (GORM) API. See how easy it is to create relationships between tables, enforce data validation rules, and change relational databases in your Grails applications.

Last month's inaugural *Mastering Grails* article introduced you to a new Web framework named Grails. Grails embraces modern practices like the Model-View-Controller separation of concerns and *convention over configuration*. Couple these with its built-in scaffolding capabilities, and Grails lets you get an initial Web site up and running in mere minutes.

This article focuses on another area that Grails makes easy: persistence using the Grails Object Relational Mapping (GORM) API. I'll start by describing what an object-relational mapper (ORM) is and how to create a one-to-many relationship. Next, you'll learn about data validation, ensuring that your application isn't plagued with *garbage in/garbage out* syndrome. You'll see the Grails ORM domain-specific language (DSL) in action, which allows you to fine-tune how your POGOs (plain old Groovy objects) are persisted behind the scenes. Finally, you'll see how easy it is to swap out one relational database for another. Any database with a JDBC driver and a Hibernate dialect is fair game.

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.

ORMs defined

Relational databases have been around since the late 1970s, yet software developers still struggle with getting data in and out of them effectively. Today's software isn't based on the relational theory that most popular databases use; it's based on object-oriented principles.

A whole class of programs — ORMs — has sprouted up to ease the pain of moving data back and forth between databases and your code. Hibernate, TopLink, and the Java Persistence API (JPA) are three popular Java APIs that tackle this problem (see Resources), although none is perfect. This problem is so persistent (no pun intended) that it even has its own catchphrase: the *object-relational impedance mismatch* (see Resources).

GORM is a thin Groovy facade over Hibernate. (I guess "Gibernate" doesn't roll off the tongue as easily as "GORM.") This means that all your existing Hibernate tricks still work — for example, HBM mapping files and annotations are both fully supported — but this article focuses on the interesting capabilities GORM brings to the party.

Object-oriented databases and Grails
Some developers want to eliminate the object-relational impedance mismatch issue by using databases that support objects natively. Ted Neward's developerWorks series *The busy Java developer's guide to db4o* tackles this subject well, showing a modern object-oriented database in action. It'd be great to see some enterprising developer write a db4o plug-in for Grails, proving that object-oriented databases are just as ready for prime time as their relational counterparts. In the meantime, using GORM and a traditional relational database is your best persistence strategy with Grails.

Creating one-to-many relationships

It's easy to underestimate the challenge of saving your POGOs to database tables. Indeed, if you are mapping one POGO to one table, things are pretty easy — the POGO properties match up perfectly with the table columns. But when you add the

slightest bit of complexity to your object model, such as having two POGOs that are related to each other, things can quickly get more difficult.

For example, take a look at the Trip Planner Web site you started in last month's article. Not surprisingly, the `Trip` POGO plays an important role in the application. Open grails-app/domain/Trip.groovy (shown in Listing 1) in a text editor:

Listing 1. The Trip class

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

Each of the attributes in Listing 1 maps cleanly and easily to a corresponding field in the `Trip` table. Recall from the last article that all POGOs stored in the grails-app/domain directory get a corresponding table created automatically when Grails starts up. By default, Grails uses the embedded HSQLDB database, but by the end of this article, you'll be able to use any relational database you like.

Travel often involves a flight, so it makes sense to create an `Airline` class (shown in Listing 2) as well:

Listing 2. The Airline class

```
class Airline {
  String name
  String url
  String frequentFlyer
  String notes
}
```

Now you want to link these two classes. To schedule a trip to Chicago on Xyz Airlines, you represent that in Groovy the same way you would in Java code — by adding an `Airline` property to the `Trip` class (see Listing 3). This technique is called *object composition* (see Resources).

Listing 3. Adding an Airline property to the Trip class

```
class Trip {
  String name
  String city
  ...
  Airline airline
}
```

That's all fine and good for the software model, but relational databases take a slightly different approach. Every record in the table has a unique ID called the *primary key*. Adding an `airline_id` field to the `Trip` table allows you to link one record to the other (in this case, the "Xyz Airlines" record to the "Chicago trip" record). This is called a *one-to-many* relationship: one airline can have many trips associated with it. (You can find examples of one-to-one and many-to-many relationships in the online Grails documentation; see Resources.)

This proposed database schema has just one problem. You may have successfully *normalized* the database (see Resources), but now the columns in the table are out of sync with your software model. If you replace the `Airline` field with an `AirlineId` field, then you've let an implementation detail (the fact that you're persisting your POGOs in a database) *leak* up to the object model. Author Joel Spolsky calls this the *Law of Leaky Abstractions* (see Resources).

GORM helps mitigate the leaky-abstraction problem by allowing you to represent your object model in a way that makes sense in Groovy. It handles the relational database issues for you behind the scenes, but as you'll see in just a moment, you can easily override the default settings. Rather than being an *opaque* abstraction layer that hides the database details from you, GORM is a *translucent* layer — it tries to do the right thing out of the box, but it doesn't get in your way if you need to customize its behavior. This gives you the best of both worlds.

You've already added an `Airline` property to the `Trip` POGO. To complete the one-to-many relationship, add a `hasMany` setting to the `Airline` POGO, as shown in Listing 4:

Listing 4. Creating a one-to-many relationship in Airline

```
class Airline {
  static hasMany = [trip:Trip]

  String name
  String url
  String frequentFlyer
  String notes
}
```

The static `hasMany` setting is a Groovy hashmap: the key is `trip`; the value is the `Trip` class. If you want to set up additional one-to-many relationships with the `Airline` class, you can put a comma-delimited list of key/value pairs inside the square brackets.

Cascading deletes

As the model stands now, you could end up with orphans in the database: deleting an `Airline` will leave `Trip` records that point back to a nonexistent parent. To avoid this scenario, you can add a corresponding `static belongsTo` hashmap in the many-side class.

Now create a quick `AirlineController` class (shown in Listing 5) in grails-app/controllers so that you can see your new one-to-many relationship in action:

Listing 5. The AirlineController class

```
class AirlineController {
  def scaffold = Airline
}
```

Recall from the last article that `def scaffold` tells Grails to create the basic `list()`, `save()`, and `edit()` methods dynamically at run time. It also tells Grails to create the GroovyServer Page (GSP) views dynamically. Be sure that both `TripController` and `AirlineController` contain `def scaffold`. If you have any GSP artifacts left over in grails-app/views from typing `grails generate-all` — either a trip or an airline directory — you should delete them. For this example, you want to be sure that you're allowing Grails to scaffold both the controllers and the views dynamically.

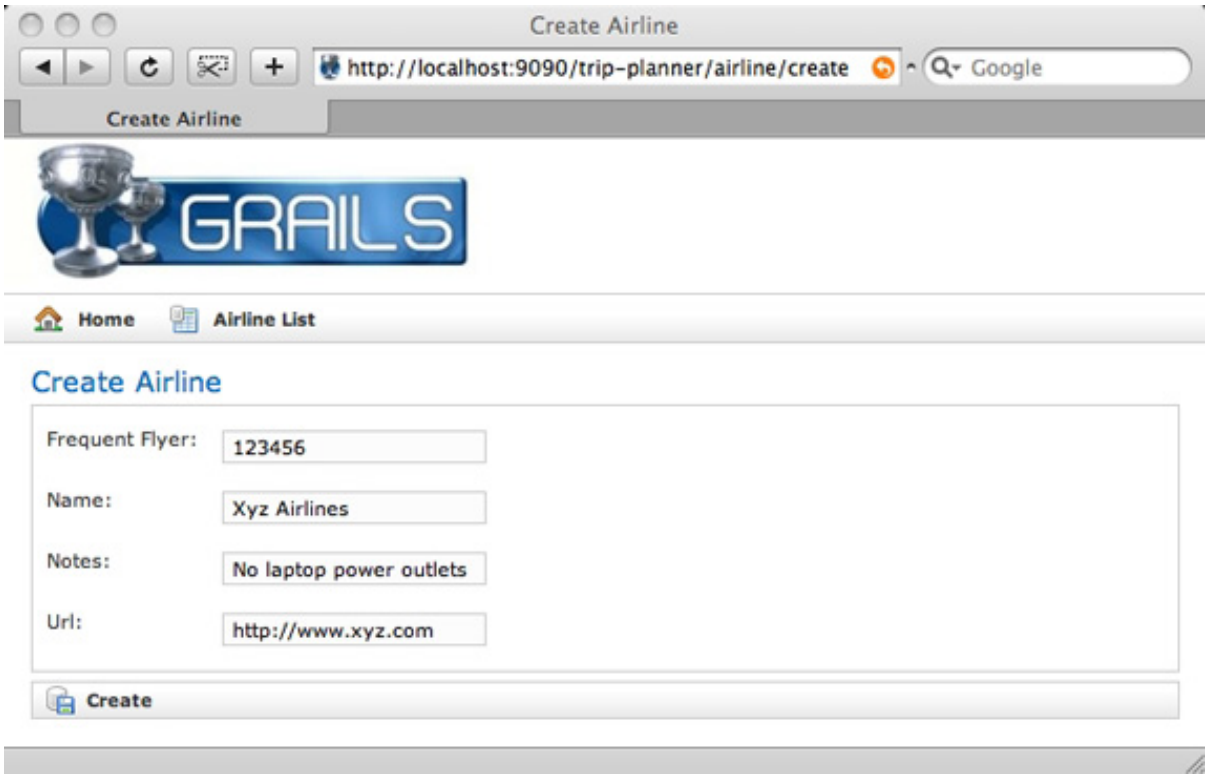
Now that your domain classes and controllers are in place, start Grails. Type `grails prod run-app` to run the application in production mode. If all goes well, you should be greeted with a message that says:

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

Avoiding port-conflict errors
If another application is already running on port 8080, see last month's article for instructions on how to change the port. I run my Grails instances on port 9090 to sidestep the problem altogether.

In your browser, you should see links for both the `AirlineController` and the `TripController`. Click on **`AirlineController`**, and fill in the details for Xyz Airlines, as shown in Figure 1:

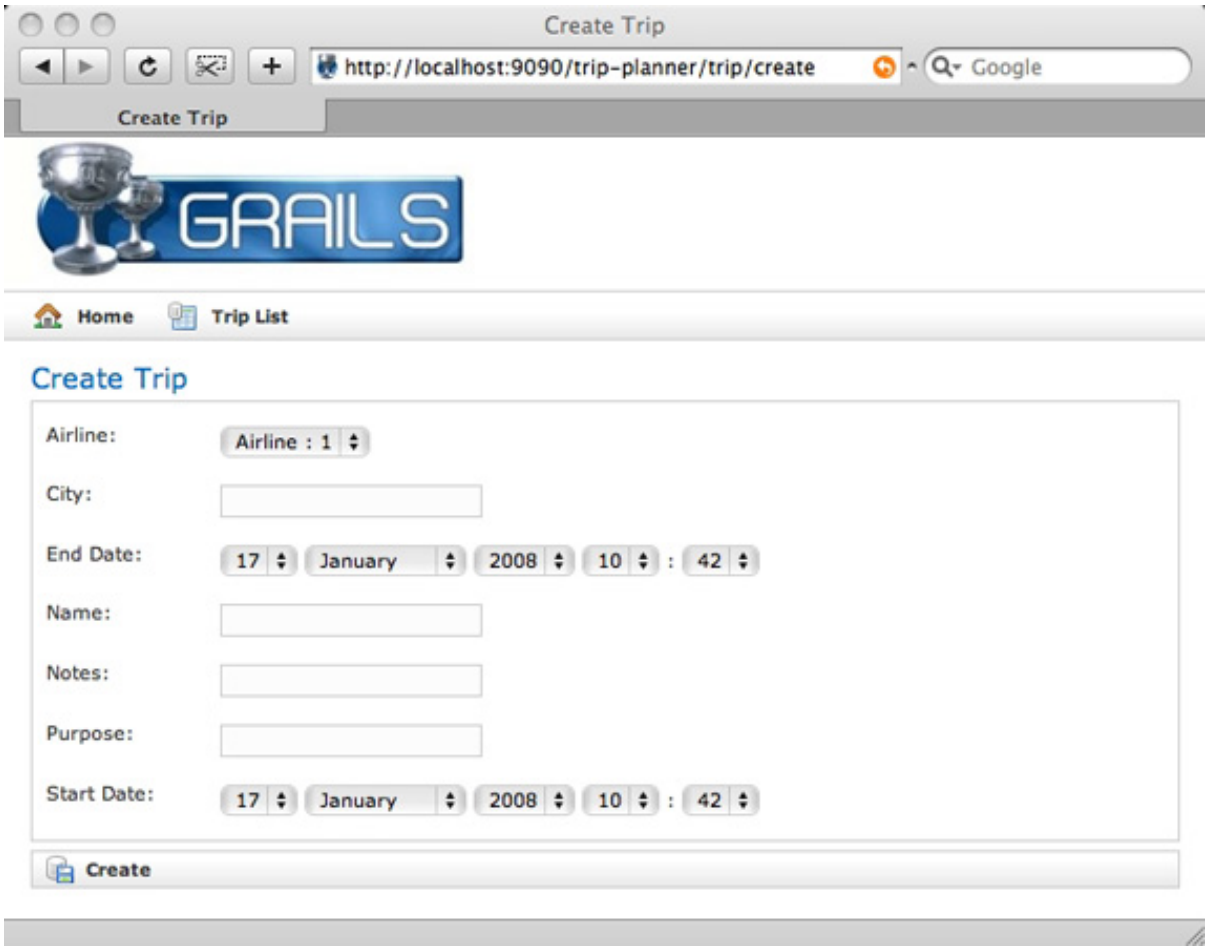
Figure 1. One-to-many relationship: The one side



Don't worry if you're not a fan of the alphabetical ordering of the fields. You'll override that in the next section.

Now create a new trip, as shown in Figure 2. Notice the combo box for `Airline`. Every record you add to the `Airline` table shows up here. Don't worry about the primary key "leaking" through — you'll see how to add a more descriptive label in the next section.

Figure 2. One-to-many relationship: The many side



Naked objects

You just saw how adding a hint to the Airline POGO (static `hasMany`) affects both the way the table is created behind the scenes and the view that's generated on the front end. This *naked objects* pattern of decorating the domain object (see Resources) is used extensively throughout Grails. By adding this information directly to the POGO, you eliminate the need for external XML configuration files. Having all the information in a single place is a great boon to productivity.

For example, getting rid of the leaky primary key that shows up in the combo box is as easy as adding a `toString` method to the `Airline` class, as shown in Listing 6:

Listing 6. Adding a toString method to Airline

```
class Airline {
  static hasMany = [trip:Trip]

  String name
```

```
  String url
  String frequentFlyer
  String notes

  String toString(){
    return name
  }
}
```

From now on, the airline's name is used as the display value in the combo box. But here's the really cool part: if Grails is still up and running, all you need to do is save Airline.groovy for the changes to take effect. Create a new `Trip` in your browser to see this in action. Because the views are dynamically generated, you can quickly go back and forth between your text editor and your browser until you get the views just right — no server reboot required.

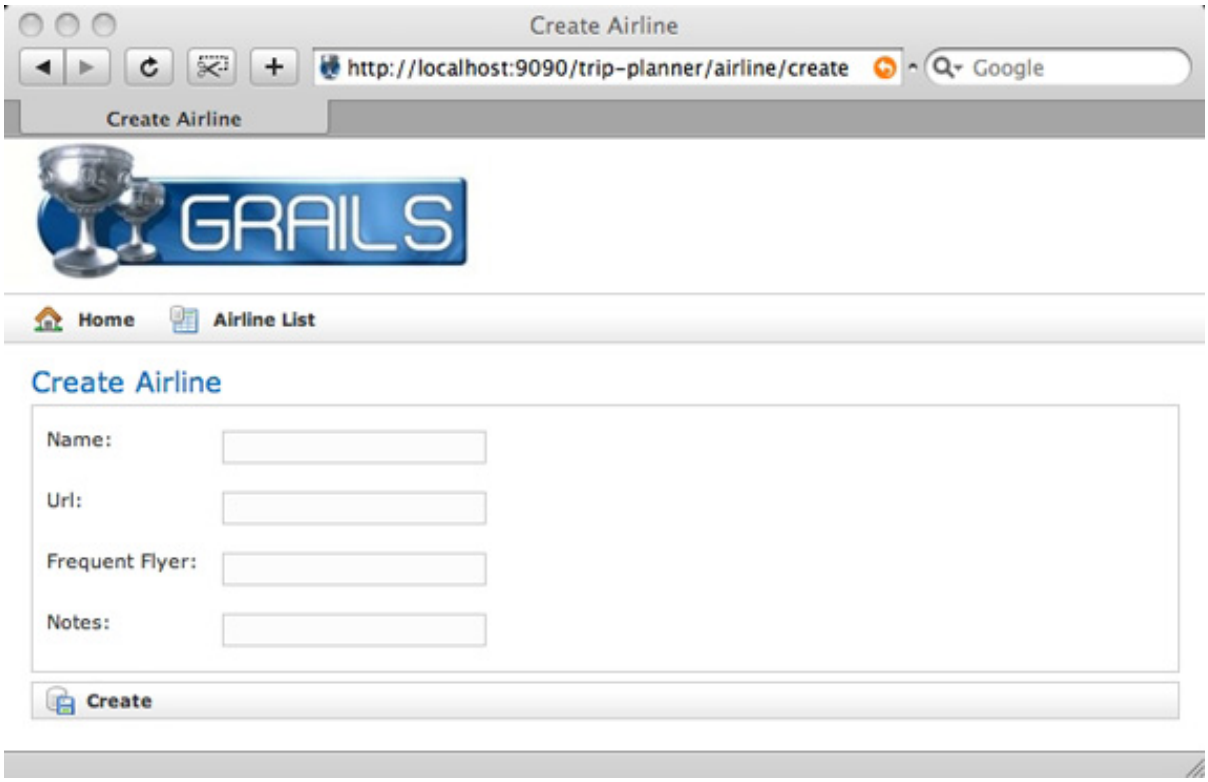
Now let's fix the alphabetical field-ordering problem. To do this, you need to add another configuration to the POGO: a `static constraints` block. Add the fields to this block in the order shown in Listing 7. (These constraints don't affect the order of the columns in the table — just the view.)

Listing 7. Changing the field order in Airline

```
class Airline {
  static constraints = {
    name()
    url()
    frequentFlyer()
    notes()
  }

  static hasMany = [trip:Trip]

  String name
  String url
  String frequentFlyer
  String notes

  String toString(){
    return name
  }
}
```

Save these changes to the Airline.groovy file, and create a new airline in your browser. The fields should now appear in the order you specified in Listing 7, as shown in Figure 3:

Figure 3. Customized field order



Before you accuse me of violating the DRY (Don't Repeat Yourself) principle (see Resources) by having you needlessly type the field names twice in the POGO, rest assured that there's a good reason for pulling them out into a separate block. Those parentheses in Listing 7's `static constraints` block won't stay empty for long.

Data validation

In addition to specifying the field order, the `static constraints` block also lets you put some validation rules in place. For example, you can enforce length restrictions on `String` fields. (They default to 255 characters.) You can ensure that `String` values match a certain pattern (such as an e-mail address or URL). You can even make fields optional or required. For a complete listing of available validation rules, see the Grails online documentation (see Resources).

Listing 8 shows the `Airline` class with validation rules added to the constraints block:

Listing 8. Adding data validation to Airline

```
class Airline {
  static constraints = {
    name(blank:false, maxSize:100)
    url(url:true)
    frequentFlyer(blank:true)
```

```
    notes(maxSize:1500)
  }

  static hasMany = [trip:Trip]

  String name
  String url
  String frequentFlyer
  String notes

  String toString(){
    return name
  }
}
```

Save this modified Airline.groovy file, and create a new airline in your browser. If you violate the validation rules, you'll get a warning, as shown in Figure 4:

Figure 4. Validation warnings

You can customize the warning messages in the messages.properties file in the grails-app/i18n directory. Notice that the default messages are localized in several languages. (See the validation section of the online Grails documentation for details on how to create custom messages on a per-class, per-field basis.)

Most of the constraints in Listing 8 affect only the view layer, but a couple affect the persistence layer as well. For example, the `name` column in the database is now 100 characters long. The `notes` field, in addition to flipping from an input field to a text area in the view (this happens for fields larger than 255 characters), flips from being a `VARCHAR` column to a `TEXT`, `CLOB`, or `BLOB` column as well. This all depends on the type of database you're using behind the scenes and its Hibernate dialect, which — it probably comes as no surprise at this point — you can override.

Grails ORM DSL

You can override the Hibernate defaults using any of the usual configuration methods: HBM mapping files or annotations. But Grails offers a third way that follows the naked-objects style. Simply add a `static mapping` block to the POGO to override the default table and field names, as shown in Listing 9:

Listing 9. Using the GORM DSL

```
class Airline {
  static mapping = {
    table 'some_other_table_name'
    columns {
      name column:'airline_name'
      url column:'link'
      frequentFlyer column:'ff_id'
    }
  }

  static constraints = {
    name(blank:false, maxSize:100)
    url(url:true)
    frequentFlyer(blank:true)
    notes(maxSize:1500)
  }

  static hasMany = [trip:Trip]

  String name
  String url
  String frequentFlyer
  String notes

  String toString(){
    return name
  }
}
```

The mapping block is especially helpful if you have existing legacy tables that you'd like to use in your new Grails application. While I just touch on the basics here, the

ORM DSL allows you to do much more than simply remap table and field names. You can override the default datatypes for each column. You can adjust the primary-key creation strategy or even specify a composite primary key. You can modify the Hibernate cache settings, adjust the fields used for foreign-key associations, and much more.

The important thing to remember is that all of these settings are consolidated in one place: the POGO.

Understanding DataSource.groovy

Everything we've done up to this point focused on tweaking the individual classes. Now let's step back and make some global changes. The database-specific configurations that are shared across all domain classes are stored in a common file: grails-app/conf/DataSource.groovy, shown in Listing 10. Pull this file up in a text editor and look around:

Listing 10. DataSource.groovy

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

The `dataSource` block is where you can change the `driverClassName`, `username`, and `password` used to connect to the database. The `hibernate` block allows you to adjust the cache settings. (No need to tweak things here unless you

are a Hibernate expert.) But the really interesting stuff happens in the `environments` block.

Recall from the last article that Grails can run in three modes: development, test, and production. When you type `grails prod run-app`, you are telling Grails to use the database settings in the `production` block. If you'd like the `username` and `password` to adjust on a per-environment basis, simply copy those settings from the `dataSource` block into the individual `environment` block and change the values. Settings in the `environment` block override those in the `dataSource` block.

The `url` setting is the JDBC connection string. Notice in `production` mode that HSQLDB uses a file-based datastore. In `development` and `test` mode, HSQLDB uses an in-memory datastore. Last month, I told you to run in `production` mode if you wanted your `Trip` records to stick around between server restarts. Now you know how to get the same thing to happen in `development` and `test` mode as well — simply copy the `url` setting from `production`. Of course, pointing Grails to DB2, MySQL, or some other traditional file-based database will solve the *disappearing records* problem as well. (You'll see settings for DB2 and MySQL in just a moment.)

The `dbCreate` value is something else that causes different behavior to occur in different environments. It is an alias for the underlying `hibernate.hbm2ddl.auto` setting, which dictates how Hibernate manages your tables behind the scenes. Setting `dbCreate` to `create-drop` tells Hibernate to *create* your tables on start-up and to *drop* them on shut-down. If you change the value to `create`, Hibernate creates new tables and alters existing tables as needed, but all records will be deleted between restarts. The default value for `production` mode — `update` — keeps all data around between restarts, as well creating or altering tables as necessary.

If you are using Grails in front of a legacy database, I highly recommend commenting the `dbCreate` value out altogether. This tells Hibernate not to touch the database schema. Although this means that you must keep the data model in sync with the underlying database yourself, it dramatically cuts down on frantic, angry e-mails from your DBAs demanding to know who keeps changing the database tables without permission.

Adding your own custom environment is easy. For example, your company might have a *beta* program. Simply create a `beta` block in DataSource.groovy next to the others. (You can also add an `environments` block to grails-app/conf/Config.groovy for non-database-related settings.) To start Grails in `beta` mode, type `grails -Dgrails.env=beta run-app`.

Changing databases

If you allow Hibernate to manage the tables for you using the `dbCreate` setting,

pointing Grails to a new database is a quick three-step process: create the database and log in, copy the JDBC driver to the lib directory, and adjust the settings in DataSource.groovy.

The instructions for creating a database and a user vary greatly by product. For DB2, there is a step-by-step tutorial that you can follow online (see Resources.) Once you have your database and user created, adjust DataSource.groovy to use the values shown in Listing 11. (The values shown here assume the database is named `trip`.)

Listing 11. DB2 settings for DataSource.groovy

```
driverClassName = "com.ibm.db2.jcc.DB2Driver"
username = "db2admin"
password = "db2admin"
url = "jdbc:db2://localhost:50000/trip"
```

If you already have MySQL installed, use the steps shown in Listing 12 to log in as the root user and create the `trip` database:

Listing 12. Creating a MySQL database

```
$ mysql --user=root
mysql> create database trip;
mysql> use trip;
mysql> grant all on trip.* to grails@localhost identified by 'server';
mysql> flush privileges;
mysql> exit
$ mysql --user=grails -p --database=trip
```

Once you have the database and user created, adjust DataSource.groovy to use the values shown in Listing 13:

Listing 13. MySQL settings for DataSource.groovy

```
driverClassName = "com.mysql.jdbc.Driver"
username = "grails"
password = "server"
url = "jdbc:mysql://localhost:3306/trip?autoreconnect=true"
```

With the database created, the JDBC driver JAR copied into the lib directory, and the values in DataSource.groovy adjusted, type `grails run-app` one more time. You now have Grails using a database other than HSQLDB.

Conclusion

That wraps up your tour of GORM. You should now have a good idea of what an ORM is, how to manage validation and table relationships, and how to swap out HSQLDB for the database of your choice.

The next article in this series focuses on the Web tier. You'll learn more about GSPs and the various Groovy TagLibs. You'll see how to break GSPs up into *partials* — fragments of markup that can be reused across multiple pages. And finally, you'll find out how to customize the default templates used in your scaffolded views.

Until then, enjoy your time *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* (Pragmatic Bookshelf, March 2008): Learn more about Groovy and Grails in Scott Davis' latest book.
- Hibernate, TopLink, and JPA: Popular Java object-relational APIs.
- Object-relational impedance mismatch: The challenges inherent in object-relational mapping.
- Object composition: Combining simple objects into more-complex ones.
- Database normalization: A technique for designing relational database tables to avoid data anomalies.
- Leaky abstraction: According to Joel Spolsky, "All non-trivial abstractions, to some degree, are leaky."
- Naked objects: Read about the architectural pattern of decorating the domain object.
- DRY: A process philosophy emphasizing that information should not be duplicated.
- "Hello World: Learn the basic features and concepts of DB2 for Linux, UNIX, and Windows" (developerWorks, December 2006): Get an overview of DB2 for Linux, UNIX, and Windows in this basic tutorial, which includes demos and hello world exercises.
- *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.
- Model-View-Controller: Grails follows this popular architectural pattern.
- 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

- DB2: Download the test drive of DB2 9.5 data server.
- 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

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.