Contents

Chapter 1: Defining the Model in GORM

The Goal of GORM

When you stop for a moment to consider what we are doing, it is really astounding. We are trying to comprehend some kind of process or system with all of its emergent and dynamic aspects, with its historical artifacts and experimental features, and to somehow encode that comprehension onto an electrically-charged platter. Most of the time, we're not just adjusting the electrical charge on the platter in front of us, though—that'd be too easy. We want to encode the information of this system into some electrically-charged platter that we have never even seen, and have only experienced by way of many, many levels of intermediaries. In short, we are communicating the behavior of an entire system to an electrically charged platter far away via an astoundingly automated game of telephone.

There seems to be two approaches when it comes time to accomplish this daunting, seemingly magical feat: the first approach begins with deep analysis of the application's domain, involves conversations where the breadth of the domain is identified and the intricacies of the relationships are explored, and finally creates the model in code (the analyst's approach); the second creates class definitions as they become relevant to functionality and relies on dynamic restructuring to handle new developments (the hacker's approach). In either case, certain qualities of GORM (Grails Object Relational Mapping) make it a very advantageous ORM technology.

For the hacker, GORM consolidates the current class definition into a single, quickly written, easily read file. That single file defines the class's properties, functionality, and relationships, which means changes to the class definition are always reflected in a single place.

For the analyst, GORM provides a very rich set of configuration options with many convenient default settings, while also letting you adjust the configuration to best fit within the envisioned system.

For either hacker or analyst, working with GORM starts the same way—by defining the model objects.

Defining Objects in GORM

Fundamentally, a GORM domain class is simply a Groovy class placed under `./grails-app/domain`. By its location, Grails infers that it is a GORM domain class and attaches to it a full set of default metadata and functionality. This metadata provides a default set of mappings from Groovy objects onto the database. These mappings are usually sufficient for development purposes and small applications. However, when the time does come for some custom configuration, metadata configuration of the GORM domain class is done by defining static properties on the class.

Creating and Defining Simple Domain Classes

To generate a simple domain class with the name "Foo," execute `grails create-domain-class Foo` in the root of your Grails application. This creates two files in your application: `./grails-app/domain/Foo.groovy` and `./test/integration/FooTests.groovy`. The first file defines the domain class, and the second provides a place to write tests for that class.

Caution When naming your classes and properties, watch out for SQL and Hibernate Query Language keywords. These keywords will cause problems when Grails tries to map your object onto the database. Should you run into this problem, the simplest solution is to customize the offending name. See the "Custom Names" section later in this chapter for more on this.

Opening up `./grails-app/domain/Foo.groovy`, we find it very barren.

Code Listing 1-1. The Foo Domain Class at Generation

```
// In ./grails-app/domain/Foo.groovy
class Foo {
}
```

Despite the emptiness of the class, quite a bit has already gone on: when the Grails application is run, this class will ensure the database matches up with the class definition and inject an entire site of querying functionality onto the class, as well as other functionality such as optimistic locking. There is a lot here, so let's open up the test file at `./test/integration/FooTests.groovy` to exercise the basics. As it sits, that file is frightfully empty.

Code Listing 1-2. The Foo Domain Class Tests at Generation

```
// In ./test/integration/FooTests.groovy
class FooTests extends GroovyTestCase {
  void testSomething() {
  }
}
```

Tip Facing down an empty test class can cause coder's block in even the most experienced developers. To get over this hurdle I always write my first test as simply `assertTrue(true)`, and then execute that to verify my testing framework is set up correctly. The next test seems a lot easier to write once there's already something on the page. This test will also catch any exceptions in the testing framework or test class setup and teardown.

Let's begin by exercising some of the functionality that Grails provides for free.

Code Listing 1-3. Demonstrating the `id` Property

```
// In ./test/integration/FooTests.groovy
void testId() {
  def foo = new Foo()
  assertNull    foo.id
  assertNotNull foo.save() // Asserts no errors
  assertNotNull foo.id     // Set in save
  assertEquals  Long, foo.id.class
}
```

Although we did not declare an `id` property, Grails has provided one for us. The property is the artificial identifier and primary key for the database row, so it is not set until the record is saved into the database with `.save()`.

Another property that Grails provides is the `version` property, which it uses to manage optimistic locking. To exercise the `version` property, though, we first need to provide a new property to change.

Code Listing 1-4. Adding the `bar` Property

```
// In ./grails-app/domain/Foo.groovy
class Foo {
  int bar
}
```

Simply declaring the property is all that it takes: Grails handles the database migration and provides reasonable default property metadata for you. In order to do this, the property must be statically typed, and of a type recognizable to Hibernate—all the primitive types (including String) are.

Now that we have a property on our domain class we can exercise the `version` property.

Code Listing 1-5. Demonstrating the `version` Property

```
// In ./test/integration/FooTests.groovy
void testVersion() {
  def foo = new Foo()
  assertNotNull foo.save()
  def version = foo.version
  foo.bar++
  assertNotNull foo.save(flush:true)
  assertEquals version+1, foo.version
}
```

The `version` property increments after each save to the database, which is how GORM implements optimistic locking. Optimistic locking is provided for free: you do not have to do anything with the `version` property except stay out of its way. You can find details on using optimistic locking in Chapter 4.

Note There is some caching going on in Code Listing 2-5. If you remove `flush:true` from the second `save` call, the test will fail. This is because calls to `save()` do not normally hit the database until the end of the transaction, but we force the save and the associated version increment by calling `save(flush:true)`.

Grails does not just provide properties and background functionality, though, it also provides an entire set of static data access methods. For more information on these, see Chapter 3.

Creating and Defining Inheritance Trees

One of the most major breaks between the relational and object-oriented paradigm is the concept of inheritance: while relational databases handle *has-a* relationships just fine via foreign keys, they do not have a clear way to mirror *is-a* relationships. This has been a pain point in many ORM technologies where an object-oriented programmer wants to define a model where behavior is inherited, but has to somehow hack a *has-a* relationship in the metadata to line things up in the database.

In Grails, extending a class is straightforward and works exactly how you would guess it would. Let's begin by executing `grails create-domain-class Product` and `grails create-domain-class Book` to create two domain classes. We will make `Book` extend `Product` through standard Java inheritance.

Code Listing 1-6. `Product` and `Book` Class Definitions

```
// In ./grails-app/domain/Product.groovy
class Product {
  String name
}

// In ./grails-app/domain/Book.groovy
class Book extends Product {
  String author
}
```

The `name` and `author` properties exist solely to demonstrate the polymorphic nature of `Book` and `Product`. These could be empty classes if you so desired. Beyond telling `Book` to extend `Product`, there is literally no further configuration needed to handle this polymorphic relationship—a polymorphic relationship is now stored in the database and reflected throughout GORM.

Code Listing 1-7. Demonstrating Domain Class Polymorphism

```
// In ./test/integration/BookTests.groovy
void testBookIsProduct() {
  assertEquals 0, Product.list().size()
  def book = new Book(name: 'Book', author: 'Author')
  assertNotNull book.save()
  assertNotNull Product.get(book.id)
  assertEquals book.name, Product.get(book.id).name
  assertEquals book.author, Product.get(book.id).author
  assertEquals Book, Product.get(book.id).class
}
```

As the test in the previous code listing shows, you can save a `Book` object and then retrieve it using the `Product` structure: the `Book` *is-a* `Product`, so queries on `Product` reflect the newly existing `Book`. Even more, the `Product` query is smart enough to return a `Book` class, so you have direct access to the `author` property.

In the background Grails is creating only a single table—`product`—which contains the attributes for both the `Product` and `Book` domain classes. In addition to the standard automatic properties (`id, version`), there is also a `class` column that specifies the class the row represents. Do not try to work with the `class` column directly: as the last assert in the previous code listing demonstrates, it is not accessible as a GORM property (it is hidden by the `Object` method `getClass()`) and simply acts as metadata.

Although this approach performs very well and behaves in intuitive ways, the single-table-per-tree approach to inheritance has a few nuances that a developer needs to be aware of. The most obvious nuance is the somewhat incongruous definition of the `author` column on the `product` table: whereas GORM defaults to defining all columns as `not null`, the `author` column is defined as nullable in the database. Despite that column

definition, the `author` field cannot be set to `null`, as the next code listing proves.

Code Listing 1-8. Demonstrating How Child Class Properties Enforce the Nullability Constraint

```
// In ./test/integration/BookTests.groovy
void testAuthorIsNotNullable() {
  def book = new Book(name: 'Book', author: 'Author')
  assertNotNull book.save()
  book.author = null
  assertNull book.save()
  assertNull book.errors.getFieldError('name')
  assertNotNull book.errors.getFieldError('author')
}
```

As long as the table is accessed via this GORM method, there is no concern about data integrity here. However, some people will be uncomfortable having a nullable column in a table that should not be able to hold the null value in the model's property.

Astute readers are already identifying a tricky problem: what about two classes in the same tree that have the same property name but different data types? GORM handles this transparently. It reuses the column but uses the more general data type. For instance, given our `Book` with a `String author` property, let's see what happens when we define a `Book2` with an `int author` property.

Code Listing 1-9. Definition of `Book2` with a Conflicting Property Type

```
// In ./grails-app/domain/Book2.groovy
class Book2 extends Product {
  int author
}
```

When this class is executed the database is left with the data type corresponding to `String`, but integer values are inserted into it. Both the `String author` and `int author` properties still work, and GORM ensures that the developer will never notice the difference, as Code Listing 1-10 demonstrates.

Code Listing 1-10. Demonstrating Coexisting Conflicting Columns

```
// In ./test/integration/Book2Tests.groovy
void testCanAddIntAuthor() {
 def book = new Book(name:'Book', author:'Author')
 def book2 = new Book2(name:'Book2', author:2)
 assertNotNull book.save()
 assertNotNull book2.save()
 assertEquals String, Book.get(book.id).author.class
 assertEquals Integer, Book2.get(book2.id).author.class
}
```

Another nuance is that every attribute for every class in the tree is in the single parent table. This means that if you have a very deep tree or a significant number of properties on child classes, you can end up with a very broad table, most of which is unused for certain classes. This is generally not a noticeable impact on performance or storage size, but it does encourage small classes and shallow inheritance trees.

If you run into a situation where you can prove that the unused columns or conflicting datatypes are a significant issue, GORM does give you another way to define inheritance trees: instead of storing all the subclasses into one very large table you can give each subclass its own table to store additional information. This eliminates any issue around unused and conflicting columns, but at the expense of creating additional tables in the database and requiring many expensive joins at fetch time. As a general rule, this alternative approach is not preferable to the single-table approach because of performance issues and behavior inconsistencies, but Grails

gives you the option to do it if you really insist. For more information on this approach, see the "Mapping Inheritance" section later in this chapter.

Adding Functionality to One Domain Class

Once you have your class defined you simply have a wrapper around the relational tables that exist in the database, perhaps with a bit of polymorphism for good measure. That model is not really object-oriented, though, until those classes get behaviors.

Custom Methods and Artificial Properties

Methods are the backbone of object-oriented programming in Groovy, and they are the defined in GORM domain classes just as they are defined for any other Groovy object. As with any other Groovy object, methods in a domain class may have either implicit or explicit return values.

Code Listing 1-11. Defining User Methods on a Domain Class

```
// In ./grails-app/domain/Foo.groovy
class Foo {
  int bar
  def returnTrue() { return !returnFalse() }
  boolean returnFalse() { return false }
}
```

Code Listing 1-12. Testing User Methods on a Domain Class

```
// In ./test/integration/FooTests.groovy
void testLogHelloAndLogGoodbyeReturnValues() {
  def foo = new Foo()
  assertTrue foo.returnTrue()
  assertFalse foo.returnFalse()
}
```

In addition to standard behavior methods, GORM also gives the user the ability to define artificial properties. Artificial properties are not declared as normal properties, but instead are declared by providing a getter and/or a setter following a particular naming convention. By defining only getters or only setters you can create read-only or write-only properties that are accessible through the standard GPath dot notation, but that are not mapped into the database in any way. As an example, in Code Listing 1-13 the `readOnly`, `writeOnly`, and `readWrite` properties are not reflected in the database and are simply convenience methods for the user of the class.

Tip A very common trick with artificial properties is using them to maintain backward compatibility with earlier versions of a class. If a property's data moves to a new name or even a different domain object, an artificial property can provide access under the old name and thereby maintain behavior. This can save you from having to update areas of code built off the previous version of the domain class.

Code Listing 1-13. Defining Artificial Properties

```
// In ./grails-app/domain/Foo.groovy
def getReadOnly() { 'This is read only' }
def setWriteOnly(String toSet) {
  /* Ignored */
}
def getReadWrite() {
  System.getProperty('eg.read.write')
}
def setReadWrite(toSet) {
  System.setProperty('eg.read.write', toSet)
}
```

The methods in Code Listing 1-13 provide the `Foo` domain class with three artificial properties: `readOnly`, `writeOnly`, and `readWrite`. Since only a getter and no setter are provided for `readOnly`, the property is read only. Since only a setter and no getter is provided for `writeOnly`, that property is write only. And since we provided both a setter and a getter for `readWrite`, it is both readable and writable. This behavior is demonstrated in Code Listing 1-14. Note that due to the dynamic nature of the Groovy language, attempting to inappropriately access a read-only or write-only property is a runtime exception and not a compile-time error.

Code Listing 1-14. Demonstrating Artificial Properties

```
// In ./test/integration/FooTests.groovy
void testArtificialProperties() {
  def foo = new Foo()
  foo.writeOnly = 'This is write only!'
  shouldFail(MissingPropertyException) {
    foo.writeOnly
  }
  assertEquals 'This is read only', foo.readOnly
  shouldFail(ReadOnlyPropertyException) {
    // Slight asymmetry, but parent class is same
    assertTrue(ReadOnlyPropertyException instanceof
                        MissingPropertyException)
    foo.readOnly = 'Trying to save read only!'
  }
  assertEquals 'This is read only', foo.readOnly
  assertNull foo.readWrite
  foo.readWrite = 'Read and write'
  assertEquals 'Read and write', foo.readWrite
}
```