Groovalicious part deux

Posted At : March 31, 2009 7:51 AM | Posted By : Jon Messer
Related Categories: Groovy

Last time left us with a couple of very simple domain objects written in Groovy. No real behavior or anything like that, and at this point they aren't anything that we couldn't have done just as easily with Transfer or even just using queries and structs. It was kind of nice not having to create any database tables, but other than that no real world benefit so far.

Here are the updated files for this post if you would like to follow along.

Even ignoring that we have no behavior in our objects (not the point of this post), what we have is less than ideal, both from the point of view of the object model and for the relational model. Employer and Employee both have 4 properties representing an address instead of just one property that is an address. Wouldn't it be better if say, we could have the address be it's own class? That way we could implement some snazzy behavior in it and not have to repeat that logic in any object that will have an Address?

Well presto-change-o here you go :

Address.groovy

```
package model

import javax.persistence.*
import model.base.*

@Embeddable
class Address
{
 @Column(name="address")
 String address

 @Column(name="city")
 String city

 @Column(name="stateCode")
 String stateCode

 @Column(name="zipCode")
 String zipCode
}
```

Cool now we have an Address class (sans behavior), but how do we get our Employer and Employee to use that class instead of just having the 4 properties? Well you'll notice the @Embeddable annotation, that essentially means that while this class is a separate class, it's data should be persisted as part of the containing class, to the database it's not an entity in it's own right. So in the classes that we want to use this new fangled Address class with what do we have to do? You guessed it, since the class is @Embeddable we declare the instance of it as @Embedded like so :

```
class Employer
{
...
 String address
 String city
 String stateCode
 String zipCode
...
}
```

Becomes :

```
class Employer
{
...
 @Embedded
 Address address
...
}
```

Now what if we change our mind and decide that an Employer should have a main address and a mailing address? Easy enough :

```
class Employer
{
...
 @Embedded
 Address address

 @Embedded
 @AttributeOverrides([
  @AttributeOverride(name="address", column=@Column(name="mailingAddress")),
  @AttributeOverride(name="city", column=@Column(name="mailingCity")),
  @AttributeOverride(name="stateCode", column=@Column(name="mailingStateCode")),
  @AttributeOverride(name="zipCode", column=@Column(name="mailingZipCode"))
 ])
 Address mailingAddress

...
}
```

The key part to note in that code snippet is that we are overriding the column names so that "malingAddress" is mapped to the address property of the Address class but is saved to the malingAddress column of the EMPLOYER table.

Excellent, we now have defined a separate class giving us granular control over our object model, but at the same time we have the data actually being persisted in the same table as the owning entity (whether by design or for legacy reasons). Now you can't really do that with any CF persistence framework that I know of, you could roll your own and do that, but who wants to write SQL or manage associations manually. So now we're getting somewhere, especially if you had some ginormous 500 column legacy table for which the object model should be implemented as 15 different classes not one class with 500 properties...

At this point our generated tables look like this :

I should note that you do actually have full control over the creation and definition of the db schema. You can use JPA like I'm using here or you could define hbm mapping files. I am leaving a lot of things out of these posts for the sake of simplicity, things that you would probably want in a real world app, like constraints, indexes, type definition (varchar(255) for every String really?). Look to the JPA docs and hibernate docs for all your options. This is also an excellent book on hibernate and ORM in general.

Anyway, you've probably seen legacy tables like the ones above, they are easy to understand, easy to ad hoc query against and performant (no need for joins) but this design is also fragile, harder to make changes to, a data integrity nightmare and it won't really scale well in terms of maintenance. So let's try something more flexible...

Let's say we wanted to have arbitrarily many addresses for an Employer and we wanted our database schema to be a little more normalized, how would we do that? Let's do it by making a new class called EmployerAddress and give Employer an association to it that is a bi-directional One To Many (why bi directional? because we can not because it makes sense in this context).

Wait we already have an address class you say, why create a new one? Well what if we wanted to add metadata to it (like maybe type and isActive). But we really don't want to have to rewrite our incredibly complex Address class or force any other classes that are using it to change. So let's have EmployerAddress just extend Address...

EmployerAddress.groovy :

```
package model
import javax.persistence.*
import model.base.*

@Entity
@Table(name="EMPLOYERADDRESS")
class EmployerAddress extends Address
{
 @Id
 @GeneratedValue
 Long employerAddressId

 String addressType
 Boolean isActive
}
```

Address.groovy becomes :

```
package model

import javax.persistence.*
import model.base.*

@Embeddable
@MappedSuperclass
class Address
{
 String address
 String city
 String stateCode
 String zipCode
}
```

We just added @MappedSuperclass before the class definition to allow these properties to be persisted with any class that extends Address (opposed to separate table), but an Address can also still be embedded like Employee is still doing. The big change comes in Employer, we get rid of the embedded addresses and add a collection of EmployerAddress objects.

```
class Employer
{

... @OneToMany(cascade=[CascadeType.ALL])
 @JoinColumn(name="employerId")
 Set<EmployerAddress> addresses

...
```

But wait you say, we have existing code that expects Employer to have an address property, not some Collection of EmployerAddress objects (and we have no units tests to catch problems)! No problem, let's add a getter and setter for address that is internally aware of the collection.

```
class Employer
{

...

 @OneToMany(cascade=[CascadeType.ALL])
 @JoinColumn(name="employerId")
 Set<EmployerAddress> addresses

 def setAddress(a){
  //clearly you would do some kind of checking here
  //to make sure a isn't already in the Set
  //and that the addressType is correct.
  def ea = new EmployerAddress(addressType:'Main Office', isActive:true)
  ea.address = a.address
  ea.city = a.city
  ea.stateCode = a.stateCode
  ea.zipCode = a.zipCode
  this.addresses.add(ea)
 }

 def Address getAddress(){
  def i = addresses.iterator();
  while(i.hasNext())
  {
```

```
   def a = i.next();
   if(a.isActive && a.addressType == 'Main Office')
   {
    return a;
   }
  }
 }

...
```

OK, so what do we have now? Here is our generated db schema :

And let's run our super sophisticated UI, with only a couple tweaks to make sure it's all good:

```
<cfimport prefix="g" taglib="/engine/tags" />

<cfscript>
 bf = application.beanFactory;
 local=structNew();
</cfscript>

<cfif not application.modelsCreated>

 Let's create some Groovy Models...<br>

 <g:script variables="#local#">
  import model.*

  variables.employer = new Employer(employees : new HashSet(), addresses : new HashSet())
  variables.employee = new Employee(surname:'Cartman', givenName:'Eric')
  variables.employer.address = new Address(city:'South Park', stateCode:'CO', address:'123 One Way Street')
  def ea = new EmployerAddress(city:'South Park', stateCode:'CO', address:'345 Two Way Road', addressType:'Mailing', isActive:true, employer : variables.employer)
  variables.employer.addresses.add(ea)

  variables.employee.address = new Address(city:'South Park', stateCode:'CO', address:'345 Dead End Lane')
  variables.employmentrecord = new EmploymentRelationship(employer:variables.employer,employee:variables.employee)

  variables.employer.employees.add(variables.employmentrecord)
 </g:script>

 Now let's jump back to CF and manipulate them... <br>

 <cfscript>
  local.employer.employerName = 'Groovy Models Inc.';
  request.hibernateSession.save(local.employmentrecord);
  application.modelsCreated = true;
  employeeArray = local.employer.employees.toArray();
 </cfscript>

<cfelse>

 Let's load our Groovy Models...<br>

 <cfquery datasource="#application.dsn#" name="local.q" maxrows="1">select employerId from EMPLOYERS</cfquery>

 <g:script variables="#local#">
  <cfoutput>
   import model.*
   variables.employer = request.hibernateSession.get(Employer.class, Long.parseLong('#local.q.employerId#'))
  </cfoutput>
 </g:script>

 <cfscript>
   employeeArray = local.employer.employees.toArray();
 </cfscript>
</cfif>

<cfoutput>
 We would suspect that #employeeArray[1].employee.surname# #employeeArray[1].employee.getGivenName()# works for #local.employer.employerName#<br>
 Let's do a query and see...<br>
</cfoutput>

 <cfquery datasource="#application.dsn#" name="seeTheyGotSaved">
  select employerName, surname, givenname
  from employers e1
   join employee_employer ee
    on e1.employerid = ee.employerId
   join employees e2
    on ee.employeeId = e2.employeeId
 </cfquery>

<cfoutput>
 And we find that #seeTheyGotSaved.givenname# #seeTheyGotSaved.surname# does indeed work for #seeTheyGotSaved.employerName#
 which has these addresses :
 <cfset i = local.employer.addresses.iterator()>
 <cfloop condition="#i.hasNext()#">
  <cfset a = i.next()>
  <br>#a.getAddress()# #a.city# #a.stateCode#
```

```
 </cfloop>
</cfoutput>

<br>
 <cfdump expand="false" var="#local#">
<br>
```

Whew, well once again it's time to kick the can down to the next post. here are the updated files for this post if you would like to follow along. Next time we'll build up a simple CF based Dao and Service for these, then we'll see how wicked fast loading millions of objects can be...