Grails Application Development

Part 6 – GORM Advanced

Ardhika

Objectives

- To learn advanced M – M mapping & Mapping strategies

Session Plan

- Hand-craft M-M
- Mapping Strategies

Mapping M-M (hide the Membership)

- Need to hand craft a lot of code
- Worth the trouble
- We are keeping the Membership class

Membership class

- Provide 2 static methods for
- Make a User member of a Circle - subscribe
- Remove User from the circle - unSubscribe

```
class Membership {
    //Constraints

  static belongsTo = [user:User, circle:Circle]

  static Membership subscribe(User user, Circle circle) {
        //Create a membership and return
  }

  static void unSubscribe(User user, Circle circle) {
       //find a membership and delete it
  }
}
```

Membership - Subscribe

```
static Membership subscribe(User user, Circle circle) {
    def subscription =
                Membership.findByUserAndCircle(user, circle)
  if(!subscription) {
     subscription = new Membership()
     user?.addToMemberships(subscription)
     circle?.addToMemberships(subscription)
     subscription.save()
  }
  return subscription
}
```

Ardhika

Membership - Unsubscribe

```
static void unSubscribe(User user, Circle circle) {
  def subscription =
                    Membership.findByUserAndCircle(user, circle)
  if(subscription) {
      user?.removeFromMemberships(subscription)
      circle?.removeFromMemberships(subscription)
      subscription.delete();
  }
}
```

User & Circle

- These classes already have a “memberships” collection
- We have used the injected methods of in Membership class
  - addToMemberships
  - removeFromMemberships
- This membership collection is not useful outside membership class
- Other classes based on membership will want to get
  - circles collection from User and
  - members(users) collection from Circle
- To maintain memberships they also need to add and remove
  - circles to/from User
  - users(members) to/from Circle

circle collection on the User class

```
def circles () {
 return this.memberships.collect {it.circle}
}

def addToCircles(Circle circle) {
 Membership.subscribe this, circle
 return circles()
}

def removeFromCircles(Circle circle) {
 Membership.unSubscribe this, circle
 return circles()
}
```

Members collection on the Circle class

```
def members() {
 return this.memberships.collect {it.user}
}

def addToMembers(User user) {
 Membership.subscribe user, this
 return this.members()
}

def removeFromMembers(User user) {
 Membership.unSubscribe user, this
 return this.members()
}
```

Consistency - Property Vs Method

- The collections are accessed using

```
def memberList = circleObject.members()
def circleList = userObject.circles()
```

- This is a method calling syntax

- Consistent behavior is to use the collections as properties

```
def memberList = circleObject.members
def circleList = userObject.circles
```

Collections as properties - add consistency

- Change in User class

```
def getCircles () {
  return this.memberships.collect {it.circle}
}
```

- Change in Circle class

```
def getMembers() {
  return this.memberships.collect {it.user}
}
```

- Now, we could access them as properties because of getter
- But GORM will try to persist them on its own as these are properties
- How to prevent it?

Use of transients Keyword

- The keyword transients is used to indicate that the variable will not take part in defining the state of an object
- That means it should not get persisted
- So, in the Circle class members need to be mentioned as

```
static transients = [‘members’]
```

- And in the User class

```
static transients = [‘circles’]
```

GORM Advanced

Mapping Strategies

Composition

- Normally one class gets mapped to one table
- At times we may define classes to represent a group of attributes
- Consider the User class

```
class User{
  Phone homePhone
  Phone workPhone
  static embedded = [‘homePhone', ‘workPhone']
}
class Phone{
  String areaCode
  String number
}
```

- Now the user table will have 4 additional columns

Composition

- If the Phone class is written in a separate Groovy file in the grails-app/domain folder you will also get a Phone table
- For embedding you need to define the Phone class inside the groovy file of User domain class
- Groovy supports multiple classes per file

Inheritance

- Base class could be abstract or concrete
- Consider (3 different domain classes/files):

```
class Task {
   String name;
}

class EffortBasedTask extends Task{
   Integer hours;
}

class ScheduleBasedTask extends Task{
   Date startDate;
   Date endDate;
}
```

Inheritance

- Grails-GORM supports 2 strategies
- Table Per Hierarchy
  - Super class and subclasses all share a single table
  - This is the default strategy
  - One table with 4 fields(name, hours, startDate & endDate)
  - Additional field as discriminator to infer the type, normally this will be the class name
  - You cannot have a non null constraint on any field except base class field - Logically!
- Table Per Subclass
  - Every subclass along with super class attributes will be mapped to a table
  - In this case there are 2 tables

Inheritance

- If you want to have full fledged validation constraints on all field go for Table Per Subclass strategy
- To do this you need to switch the default strategy off

```
class Task {
  string name;
  static mapping = {
      tablePerHierarchy false
  }
 }
```

- Overuse of inheritance and Table Per Subclass strategy may affect the performance

Ardhika

Inheritance

- Polymorphic Query

```
def tasks = Task.list() //Get all Tasks

def tasks = EffortBasedTask.list()
//Will get you only effort based tasks
```

Collections

- GORM also supports mapping of basic collection types
  - Sets
  - Lists
  - Maps
- SET
  - Set is an unordered collection that cannot contain duplicates

```
class Circle {
  static hasMany = [discussions:Discussion]
}
```

  - The discussions property that GORM injects is a `java.util.Set`
  - Sets guarantee uniqueness but not in order

Collections

- Sorted Set
  - To have custom ordering you configure the Set as a SortedSet

```
class Circle{
   SortedSet discussions
   static hasMany = [discussions:Discussion]
}
```

  - In this case a java.util.SortedSet implementation is used
  - That means you must implement java.lang.Comparable in your Discussion class

```
class Discussion implements Comparable {
   String topic
   int compareTo(obj) {
      topic.compareTo(obj.topic)
   }
}
```

Collections

- Lists
  - To keep objects in the order in which they were added and to reference them by index like an array

```
class Circle {
  List discussions
  static hasMany = [discussions:Discussion]
}
```

  - In this case when you add new elements to the discussions collection the order is retained in a sequential list indexed from 0 so you can do:

```
  circle.discussions[0]  // get the first discussion
```

  - Elements must be added to the collection before being saved

```
def discussion = new Discussion(topic:'Some topic to be
discussed', owner : someUser)
circle.addToDiscussions(discussion)
circle.save()
```

Collections

- Bags of Objects
- Don’t want uniqueness or ordering ? Go for simple Collection!
- AddTo and RemoveFrom Collections are mapped as a Bag that don't trigger to load all existing instances from the database.
- Will perform better and require less memory than using a Set or a List

```
class Circle {
  Collection discussions
  static hasMany = [discussions:Discussion]
}
```

Disambiguation of bidirectional collection Mapping

- Remember when we went from 1 - M to M - M, you were asked to comment the ownedCircles collections when we added the memberCircles?
- What would have happened if we kept it?
- When we did bidirectional 1 - M the classes were like

```
Class User {
   //constraints
   //Other fields

   static hasMany =
     [ownedCircles : Circle]

   //methods
}
```

```
Class Circle {
   //constraints
   //Other fields

   User Owner

   //methods

}
```

Disambiguation of bidirectional collection Mapping

- When we added circle objects to the ownedCircles set of the user, GORM was setting the owner field in the circle object
- If we had kept that added another collection at the user end
- static hasMany = [memberCircles : Circle]
- GORM would have got confused because on the many side (Circle) we have a variable to link to the one side (User)
- But the one side we have 2 collections, which one should be mapped to the owner?
- Take another ambiguous case
    - A circle has a owner & moderator
    - User has 2 collections ownedCircles & moderatingCircles

Disambiguation of bidirectional collection Mapping

- How to kill ambiguity?
- Use mappedBy clause to draw that arrow for GORM

```
class User {
  //other code...
  static hasMany = [ownedCircles : Circle, ]
                    memberCircles : Circle ]
  static mappedBy = [ownedCircles : "owner" ]
  //other code...
}
```

- Specify the collection and the name of the property to map to on the many side in the one side
- for the other case

```
  static hasMany = [ownedCircles : Circle, ]
                    moderatingCircles : Circle ]
  static mappedBy = [ownedCircles : "owner",
                    moderatingCircles : "moderator"]
```

Lazy - Don't be!

- Association fetching is by default Lazy!
- Associations are fetched only when needed
- Consider this

```
Circle circle = Circle.get(1)
for(discussion in circle.discussions)
    println discussion.owner.email
```

- GORM will execute
  - one query for fetching the circle
  - Another for fetching discussions
  - And for each discussion one query to fetch the owner
- This is a classic N+1 query problem
- How to avoid multiple queries?

Eager fetching

- If you want to avoid lazy fetching switch lazy off
- In Circle class specify

```
static mapping = {
   discussions lazy:false
}
```

- Now the discussions will be loaded when the circle is fetched but there will be 2 queries
- To avoid another query use

```
static mapping = {
   discussion fetch:join
}
```

- This may be a costly affair if you don't put a limit to the number of results

Eager fetching - rule of thumb!

- Fetching with join works well for single ended association
- fetching the owner along with discussion

- But improper use of eager & join fetching may potentially load all of the data

- Use eager fetching for one-many collections
- Use Join fetching for single ended associations

Batch fetching

- You can limit only n records at a time by specifying a batch size
- In Circle class specify

```
static mapping = {
   discussions batchSize:20
 }
```

- Assuming 100 discussions in a circle for the loop GORM will execute 5 queries

- Or in the Discussion class specify

```
static mapping = {
   batchSize:20
 }
```

Locking

- By default GORM (Hibernate) uses optimistic locking
- When we update Hibernate will check the version column and might throw a StaleObjectException
- To avoid this we do a programmatic check - see the update action code in any controller
- Worst alternative is to use a pessimistic locking

```
User user = User.get(1)
user.lock() //lock obtained
user.email = newEmail
user.save() //lock released
```

- Here “select for update” will be used instead of “select”
- What is somebody else update between get & lock?

Locking

- Lock while you get

```
 User user = User.lock(1) //lock obtained
user.email = newEmail
user.save() //lock released
```

- Lock while finding

```
User user = User.findById(1, [lock:true])
```

- Lock with criteria

```
User user = User.createCriteria().get {
    eq("id", 1)
    lock true
}
```

GORM Events

- You could write code to handle these events in domain classes
- We want to trim the firstName & lastName in the User class before inserting the record

```
Class User {
    //other code
    def beforeInsert() {
        firstName = firstName.trim()
        lastName = lastName.trim()
    }
}
```

- You could do the same in beforeUpdate() also
- You could update time-stamp fields (createdDate, updatedDate) and/or create audit log records

GORM Events

- Event handling methods
  - beforeInsert - Executed before an object is initially persisted to the database
  - beforeUpdate - Executed before an object is updated
  - beforeDelete - Executed before an object is deleted
  - beforeValidate - Executed before an object is validated
  - afterInsert - Executed after an object is persisted to the database
  - afterUpdate - Executed after an object has been updated
  - afterDelete - Executed after an object has been deleted
  - onLoad - Executed when an object is loaded from the database

You could save an object inside these (risky) but never flush since this would trigger a recursion resulting in stack overflow!

Timestamps

- You dont have to this in GORM events actually
- Declare 2 properties of Date type in your domain class

```
Date dateCreated //insert time
Date lastUpdated //update time
```

- GORM will automatically fill this for you when ever you save

- For some reason you want to have these properties but dont want GORM to fill them for you
- Specify this in the domain class

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

Ardhika

Default Sorting

- You could supply parameters to sort while you fetch records using list or listBy methods
- def circles = Circle.list(sort : ‘name’)

- Or specify this in Circle class

```
static mapping = {
    //other mappings
    sort name : “desc”
}
```

- You could sort the associations also - in Circle class

```
static mapping = {
    discussions sort:”topic” order:”desc”
}
```

GORM Advanced

Configuring Databases

Datasource configuration

- Look at config/Datasource.groovy
- Consumed by ConfigSlurper
- Configurations for
- A datasource at higher level
- Hibernate properties
- Datasource customization for 3 environments
  - Development
  - Test
  - Production
- Using HSQL DB (memory & file)
- Needs knowledge for setting up JDBC connection

Datasource configuration

- dbCreate Property
  - create-drop - every time you run the app DB is created and dropped
  - update - keeps tables and data intact
  - create - keeps tables but deletes data in every run
- update option needs to be used with caution for drastic changes in DB design or use db migration plugin - refer guide
- Can configure multiple datasources - refer guide
- For development use create-drop and use the bootstrap to setup your test data

Configuring other Databases

- How to configure for MySQL, Oracle etc. ?
- update driverClassName, url, username & password
- Check for the correct dialect setting in Hibernate section

- Copy the JDBC driver in “lib” folder
- Or
- Setup the dependency using ivy and fetch with Maven - refer guide

Thank You!

Ardhika

Bala Sundarasamy
bala@ardhika.com