Grails Application Development

Part 2 – GORM
Grails Object Relational Mapping

Objectives

- To learn to use GORM for designing and implementing domain objects for Grails

Session Plan

- Introduction to GORM
- Simple Mappings

GORM - Introduction

- GORM is built on top of Hibernate
- Provides mapping syntax (DSL) for
  - One-One
  - One-Many
  - Many-Many
- Controls the way code generation /scaffolding works
- Injects persistence and retrieval methods on domain objects
- Uses conventions like
  - Class name -> table name
  - Attribute names -> column name
- Injects ID (surrogate Key) and version attributes for managing concurrent updates

CRUD operations

- Now that we have a User class, let us learn methods that are provided (injected by GORM)
- Creating an instance

```
def user = new User(firstName:'Tim', lastName:'Norton',
                    userName:'timnor', password:'hello123',
                    email:'tim@norton.com')
user.save() //save method injected by GORM
```

- When we create a new User object it will be in transient state
- After we save, the object will become persistent and any update we do with in the same transaction will reflect in the database

```
user.email = 'tim@norton.co.uk' //no need to call save again
```

CRUD – Reading User(s)

- When we create an instance and save, the id field gets populated
- We could retrieve a User object from the database using

```
def user = User.get(1) //using the id
println user
```

- To get all the Users from the Database

```
def users = User.list()
users.each { println it }
```

- To ensure certain user exists in the database

```
if (User.exists(1) ) {  //check using the id
    //code here
}
```

CRUD - Update

- When you retrieve an object from the database and keep it beyond the transaction boundary, the object gets detached
- When we modify this object we need to explicitly call save() again or update()

```
user.email = 'tim@norton.co.uk'
user.save()  //persist the instance
```

- Normally the changes made to the object even after save() is called will not be persisted immediately. Happens only at the end of thread life time (transaction end)
- To persist as soon as save() is called use

```
save(flush:true)
```

CRUD - Delete

- To delete an object (User) call

```
def user = User.get(1)
user.delete()
```

- Like update deletes also happen at the end of thread life time / transaction end
- To make delete happen in place use flush parameter like save method

```
def user = User.get(1)
user.delete(flush:true)
```

GORM - Grails Object Relational Mapping

Data Retrieval & Finders

Retrieval of Data

- There are other retrieval mechanisms
  - Dynamic finders
  - Query By Example
  - Criteria query etc.

Retrieval

- To retrieve objects without any conditions we use list() method
- Comes with parameters (used for pagination)
  - max: The maximum number of instances to return
  - offset: The offset relative to 0 of the first result to return
  - sort: The property name to sort by
  - order: The order of the results, either asc or desc for ascending and descending order respectively for the sort

```
User.list() //All Users
User.list(max:10, sort: 'firstName', order: 'asc')
```

- OrderBy - listOrderBy<fieldname>()

```
User.listOrderByLastName()
```

  - Dynamic method use with any field

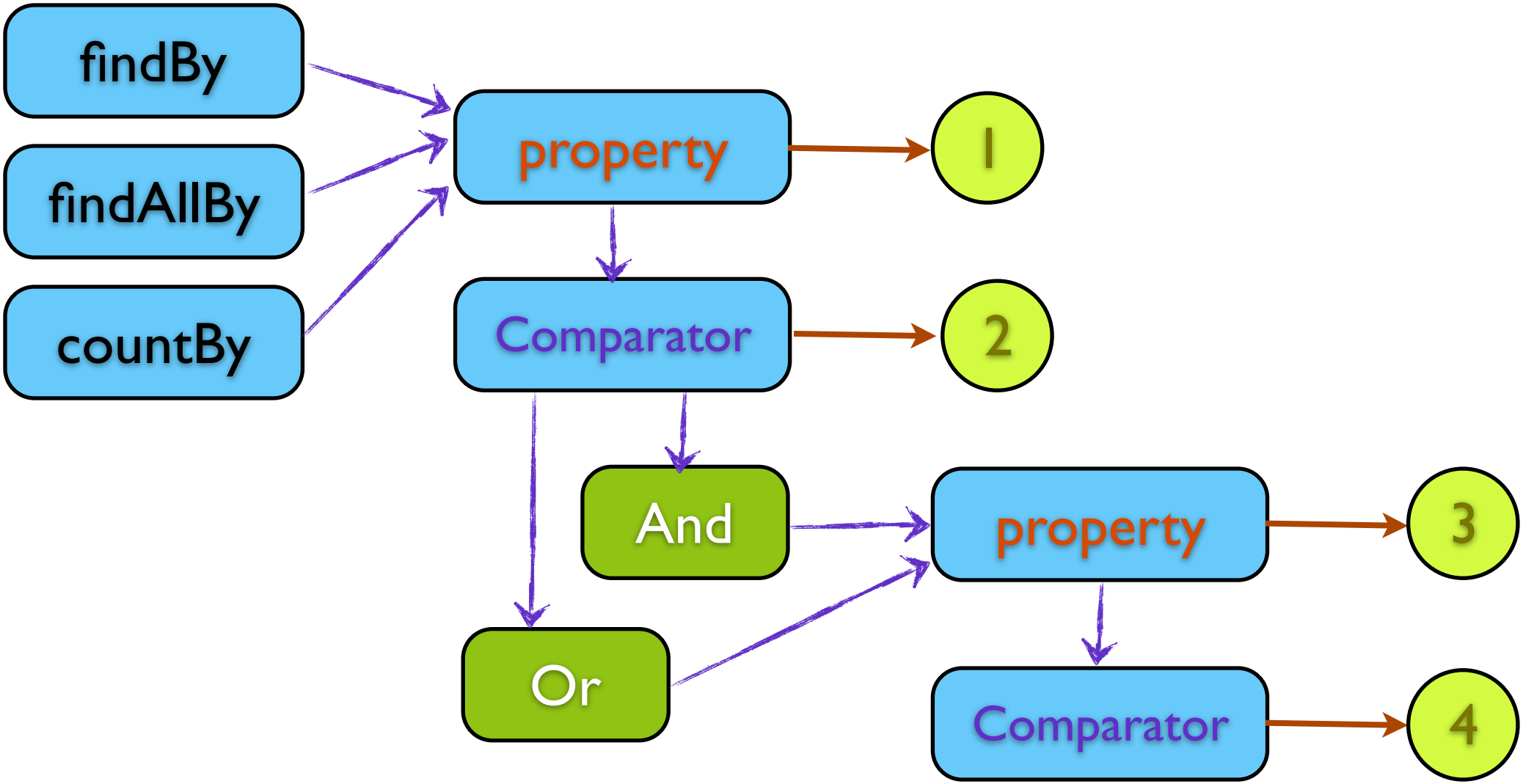
Dynamic Finders

- GORM injects finder methods to the domain classes based on the function call
- You need to specify the call conforming to some syntax and the magic works

```
User.findByFirstName('Peter')   //Exact match
User.findByFirstNameLike('Pet%') //Begins with Pet
User.findByFirstNameIlike('pet%') //Ignore case
```

- The above methods fetch the first match from the table
- Another variant findAllBy will fetch all the matches as List
- There is a method countBy which provides the count of matches
- You cannot use them beyond 2 fields / properties!

Dynamic Finders



Dynamic Finders - Comparators

Comparator	Params	Sample
LessThan	1	findByAgeLessThan(40)
LessThanEquals	1	FindByAgeLessThanEquals(40)
GreaterThan	1	findByAgeGreaterThan(30)
GreaterThanEquals	1	findByAgeGreaterThanEquals(30)
Like	1	findByFirstNameLike('Pet%')
Ilike	1	findByFirstNameIlike('Pet%')
IsNull	0	findByHomePageIsNull()
IsNotNull	0	findByHomePageIsNotNull()
NotEqual	1	findByFirstNameNotEqual('Peter')
Between	2	findByAgeBetween(30, 40)

QBE – Query By Example

- Use find method by passing a sample object with matching fields as parameter
- To find Users with same last name

```
User.find(new User(lastName:'Cook'))
```

- To find Users with same first name

```
User.find(new User(firstName:'Peter'))
```

Criteria Query

- You can build a complex filter conditions using Criteria
- Grails provides a DSL to build the criteria over HibernateCriteriaBuilder
- Very useful when filtering is needed on more than 2 fields

```
def criteria = User.createCriteria()
def googlers = criteria.list {
    or {
        ilike('homePage', '%blogspot%')
        ilike('email', '%gmail%')
    }
}
```

Criteria Query

- Short-hand form

```
def googlers = User.withCriteria {
    or {
        ilike('homePage', '%blogspot%')
        ilike('email', '%gmail%')
    }
}
```

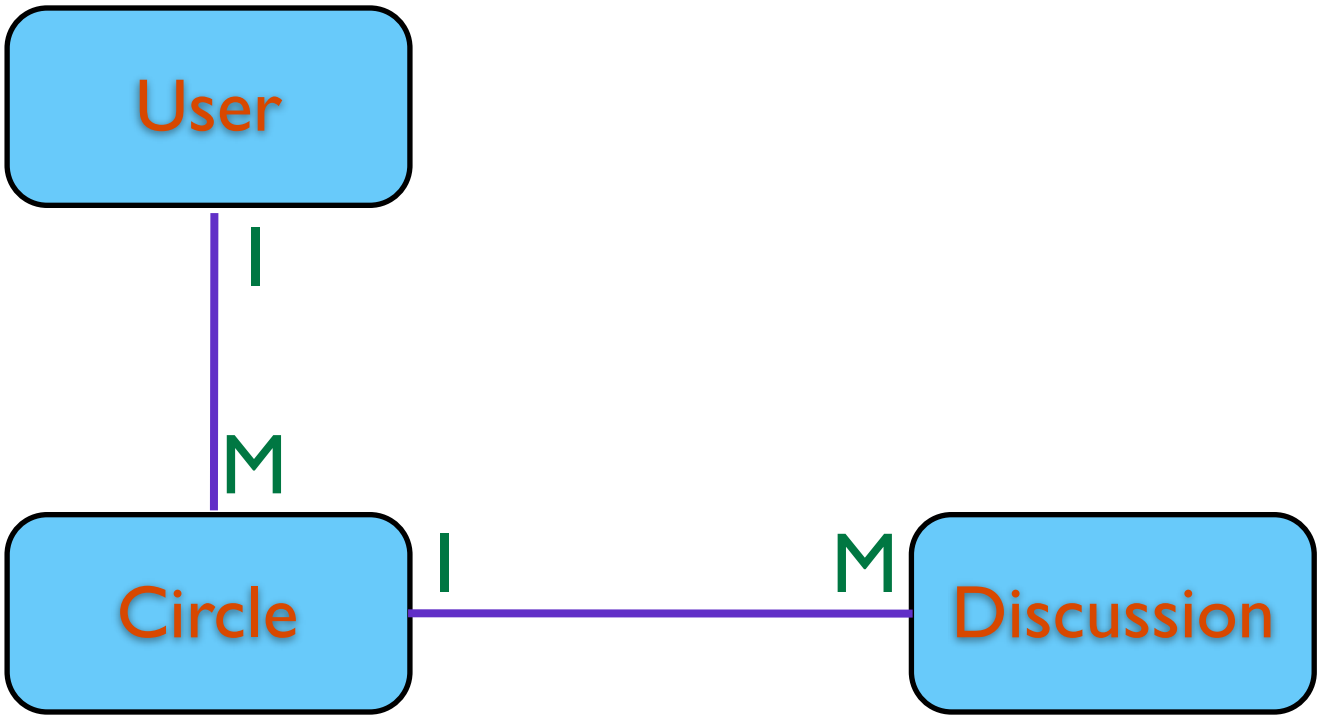
- For a full list of comparators and operators see Grails reference in the doc folder of Grails installation

GORM - Grails Object Relational Mapping

Basic Mapping

Mapping one-to-many (1-M)

- A User can create Learning Circles
  - User object will have many circles created by him/her
- A Circle can have many discussions



Mapping 1 - M

- User - Circle

```
class User {
    //Constraints
    //Fields
    static hasMany=[ownedCircles:Circle]
    //Methods
}
```

```
class Circle {
    static constraints = {
      name blank:false, size:10..50
      description blank:false, size:10..250
      owner nullable:false
    }
   String name
   String description
   User owner
}
```

Mapping 1 - M (User - Circle)

- Bidirectional 1 to many
- User will have a property of type Set by name ownedCircles
- Circle has an attribute named owner (User type)
- When we load (using get or finder) a User object the dependent objects (circles created by User) are loaded lazily (only when the property is accessed) – More select(s) fired
- Save and updates are cascaded from User to Circle
- But not delete (when a User is deleted Circles are not deleted)

Mapping 1 - M (User - Circle)

- With a Set (circles) there are 2 methods injected into User class
- To add a Circle under a User

```
userObject.addToOwnedCircles(newCircleObject)
```

- To remove a Circle from a User

```
userObject.removeFromOwnedCircles(aCircleObject)
```

- You can access the circles owned by a User

```
def user = User.get(1)
user.ownedCircles.each { println it }
```

Mapping 1 - M

- Circle - Discussion

```
class Circle {
    //Constraints
    //Fields
    static hasMany=[discussions:Discussion]
    //Methods
}
```

```
class Discussion {

    static constraints = {
      topic blank:false, size:15..250
    }

   static belongsTo=[circle:Circle]

   String topic //need to add more
}
```

Mapping 1 - M (Circle - Discussion)

- Bidirectional 1 to many
- Circle will have a Set by name discussions
- Discussion belongs to a Circle and holds Circle object in a variable by name circle
- In this case also the dependent objects are loaded lazily (only when the attributes are accessed) – More select(s) fired
- Save and updates are cascaded from Circle to Discussion
- When a Circle object is deleted discussions belonging to that Circle are also deleted due to the belongsTo clause

Mapping 1 - M (Circle - Discussion)

- With a Set (discussions) there are 2 methods injected into Circle
- To add a Discussion to a Circle

```
circleObject.addToDiscussions(newDiscussionObject)
```

- To remove a Discussion from a Circle

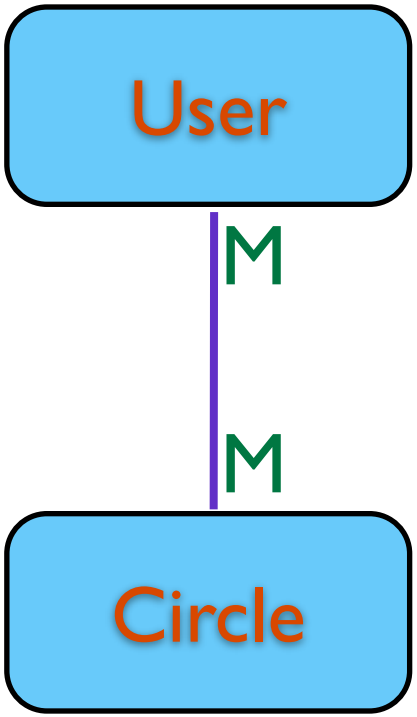
```
circleObject.removeFromDiscussions(aDiscussionObject)
```

- You can access the discussions of a Circle by

```
def circle = Circle.get(1)
circle.discussions.each { println it }
```

Mapping - many to many (M-M)

- Users can become members in Circles
- One User can become a member in many circles
- A Circle can have many users as members



- Already we have a 1 - M relationship between user and circles
- That is ownership relation

- Here it is membership relation

Mapping M - M

- User - Circle in a membership relation
- For this exercise we will remove the ownership relation from the User class for time being

```
class User {
    //Constraints
    //Fields
    //static hasMany=[ownedCircles:Circle]
      static hasMany=[memberCircles:Circle]
      static belongsTo = Circle //What???
    //Methods
}
```

```
class Circle {
    //Constraints
    //Fields
    static hasMany=[discussions:Discussion, memberUsers:User]
    //Methods
}
```

Mapping M-M (User - Circle membership)

- User will have a Set by name memberCircles
- Circle will have a Set by name memberUsers
- You can manage dependency from only one side (Circle side) because User has a belongsTo attribute set to Circle
- In this case also the dependent objects are loaded lazily
- There is another table created to manage the relationship between User and Circle
  - When a Circle is deleted the association is dropped with user
  - Deleting a User is not allowed if there is an association with Circle because User is an owned entity

Mapping M-M (User - Circle membership)

- With a Set (memberUsers) 2 methods injected into Circle
- To add (enroll) a user to a Circle

```
circleObj.addToMemberUsers(userObject)
```

- To remove a user from a Circle

```
circleObj.removeFromMemberUsers(userObject)
```

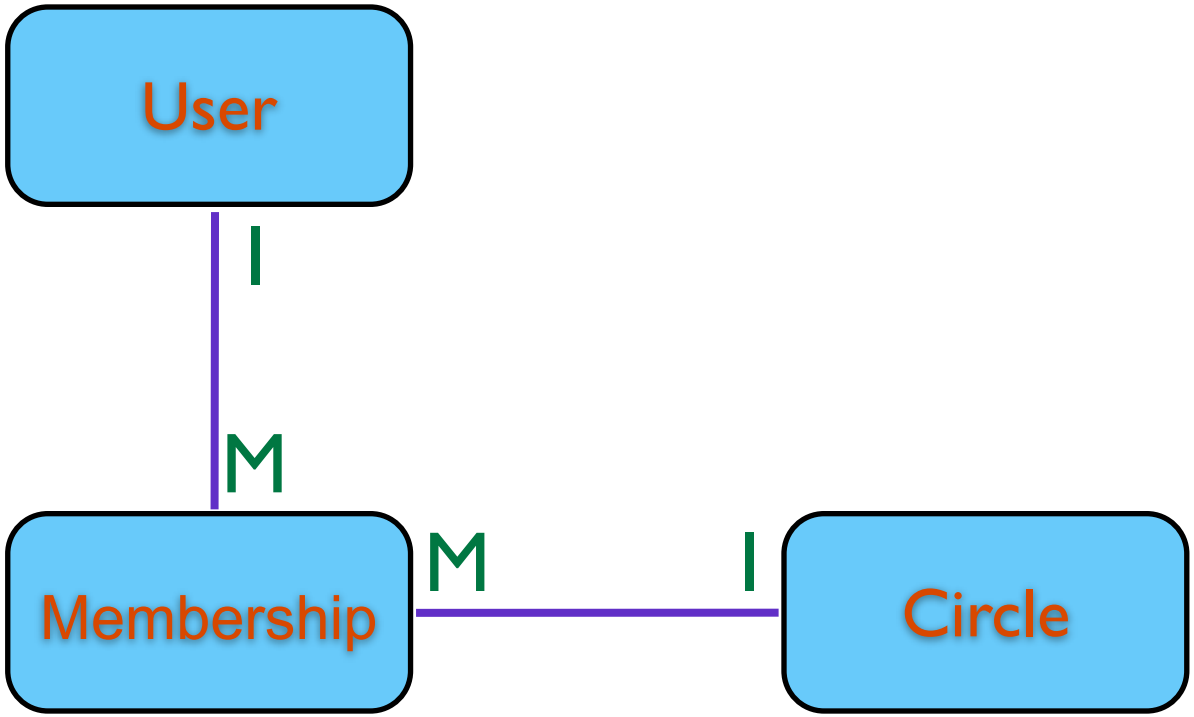
- Methods similar to above are available in User also
- `addToMemberCircles` and `removeFromMemberCircles`

Mapping M-M (User - Circle membership)

- Scaffolded code does not handle M – M relationship properly
- When a User is deleted the association with Circle is not dropped
- But when a Circle is deleted the associations are dropped
- Solution
  - It is better to break an M-M relationship into a 1 – M & M – 1 relationship by creating a 3rd domain class

Breaking M - M

- Create a domain class by name Membership



Breaking M - M

```
class Membership {

    static constraints = {
    user nullable:false
    circle nullable:false
    }

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

Breaking M - M

- Change the following line in Circle domain class

```
static hasMany=[discussions:Discussion, memberUsers:User]
```

- Into

```
static hasMany = [discussions:Discussion,
                  memberships:Membership]
```

- From User class drop these lines

```
static hasMany=[memberCircles:Circle]
    static belongsTo = Circle //What???
```

- bring ownership relation back this back and add the membership collection

```
static hasMany=[ownedCircles:Circle, memberships:Membership]
```

- Create controller for Membership with scaffold and run!

Breaking M - M

- Membership controller/UI not intuitive to the user
- Selecting Circle and User through drop downs for creating combinations (Memberships)
- We see memberships (not just users enrolled) in the Circle and User screens
  - Bad without toString() for Membership class

Please add toString method to all domain classes

- Will we be able to hide the third domain object (Membership) from the users completely?
- And provide a collection of Users on the Circle end & circles on the User end?

Mapping M-1

- M – 1 is most trivial & simple
- Unidirectional version of 1 – 1
  - Every Discussion is started by a User who owns it
- But a User can start many discussions (M-1)
- Or many discussions can be started by one User

Mapping M-1

```
class Discussion {
    static constraints = {
     topic blank:false, size:15..250
    }
  static belongsTo=[circle:Circle]
  String topic
  User owner
}
```

- But in User class there is no mention of Discussion
- This will make the relation unidirectional
- If you add a discussions attribute of type Discussion in the User, this will become the usual 1-M relationship

```
static hasMany = [discussions : Discussion]
```

- A discussion collection is more appropriate at Circle not User

Thank You!

Bala Sundarasamy
bala@ardhika.com