GORM II

Getting more control

At The End of This Session

- You will understand different ways to query for data
- You will see how GORM maps relationships to the database
- You will be able to customize how GORM maps to the database, to work with existing tables and organization conventions

Finding the Right Data

- Basic CRUD Review
- Criteria queries
- Domain.where()
- Named queries (detached criteria)

Review - Example Domain

- Buffer Overflow application
- Domain classes:
  - Question
  - Answer
  - User
- We'll make use of the dbconsole as well.
- grails-training - advancedGormStart branch
  - Has example sources
  - Integration test for queries as well

Add another property to Question

```
class Question {
    User user
    String title
    String text
    boolean deleted = false

    Date dateCreated
    Date lastUpdated

    static hasMany = [answers: Answer]
    static constraints = {...}
}
```

Queries with Basic CRUD (Review)

```
// If you know the ID
Question question = Question.get(55)

// List all, or paginate
def questions = Question.list()

User author = User.findByUsername("bmarley")

def question = Question.findByUser(author)
```

Criteria Queries - 1

- Uses a Groovy builder to create a query
- Can be used with *createCriteria* or *withCriteria* methods

```
User author = User.findByUsername("bobmarley")

// Find questions created by the user.
def qc = Question.createCriteria()
def questions = qc {
    eq("user", author)
}
```

Criteria Queries - 2

```
def qc = Question.createCriteria()
def questions = qc {
    or {
        eq("user", author)
        answers {
            eq("user", author)
        }
    }
    eq("deleted", false)
}
```

Criteria Queries - 3

```
def qc = Question.createCriteria()
def questions = qc {
    or {
        eq("user", author)
        answers {
            eq("user", author)
        }
    }
    order("lastUpdated", "desc")
}
```

Where Queries

```
def query = Question.where {
    user == author && deleted == false
}

def questions = query.list()
```

Combining Where Queries

```
def notDeleted = Question.where {
    deleted == false
}
def query = notDeleted.where {
    user == author
}

def questions = query.list()
```

Where Queries - Operators

Operator	Description
==	Equal To
!=	Not Equal To
>	Greater Than
<	Less Than
>=	Greater Than or Equal
<=	Less Than or Equal
in	Contained Within the Given List
==~	Like a Given String
=~	Case Insensitve Like

Named Queries - 1

Named, or detached queries

- Let you define a criteria and use the query again and again.
- Can specify a number of ways, often as a namedQueries static block on a domain class

Named Queries - 2

```
class Question {
    static namedQueries = {
        querySinceLastLogin { lastLogin ->
            fetchMode "answers",
                 FetchMode.JOIN
            eq("deleted", false)
            gt("lastUpdated", lastLogin)
        }
    }
}
```

Fetch Mode

Grails suggests using JOIN for single-ended associations and SELECT for one-to-many

JOIN for one-to-many will most likely end up with duplicate results

Be careful how and where you use eager loading, you could end up with your entire db in memory

Calling Named Queries

Pass values to the query, then call a method such as list() or count():

```
def questions =
     Question.querySinceLastLogin(lastLoggedIn).list()

def count =
     Question.querySinceLastLogin(lastLoggedIn).count()
```

Projections

Projections allow you to query for more than just domain objects:

```
def qc = Question.createCriteria()
def max = qc.get {
        eq('user', author)
        projections {
            max('dateCreated')
        }
 }
```

Querying with HQL

```
 def results = Question.executeQuery(
       "select q.user from Question q, Answer a"
        + " where q.user = a.user"
        + " group by a.user")

def results = Question.findAll(
       “from Question where username = “bobmarley”)
```

Relating Tables

- Using other collection types
- Performance problems with sets and lists

Add and Remove Related Objects

With hasMany, you have:

```
question.addToAnswers(answer)
question.addToAnswers(text:'Text', user:user)

question.removeFromAnswers(answer)
```

Other Collection Types

- Default is Set
  - No duplicates allowed
- You can specify other collection types:
  - SortedSet - keep objects in sort order
  - List - keep objects in insert order
  - Collection - store in a Bag

When using a list, must add first then save:

```
answer.addToQuestions(question)
question.save()
```

Other Collection Types - Example

```
class Question {
    static hasMany = [answers: Answer]
    List answers
}
```

Performance Problems

Hibernate needs to load entire List into memory. Can be a performance problem.

Can manually create a join table

- UserRole is very common
- Must manually maintain linkage

Defining How GORM Maps to DB

- Using constraints to control type mapping
- Defining table and column names
- Adding an index
- Controlling join tables
- Changing ID generation
- Timestamp versioning
- Controlling mapping globally

Using Constraints

Constraints do more than validation.

Can also impact schema generation

- maxSize, size, inList on Strings
- max, min, range, scale on numeric types

Default string sizes vary

- Always specify size for strings
- Often default to 255 characters

Defining Table and Column Names

```
class Question {
  User user
...
  static mapping = {
    table 'question_authority'
    user column: 'author_id'
  }
}
```

Adding An Index

```
class Question {
  User user
...
  static mapping = {
    table 'question_authority'
    user column: 'author_id', index: 'AuthorIdx'
  }
}
```

Controlling the Join Table

```
class Question {
  static hasMany = [answers:Answer]
...
  static mapping = {
    table 'question_authority'
    answers joinTable: [name: 'question_answ',
        key: 'question_id',
        column: 'answer_id'
      ]
  }
}
```

Changing ID Generation

```
class Question {
    ...
    static mapping = {
        table 'question_authority'
        id generator: 'hilo',
            params: [table: 'hi_value',
                     column: 'next_value',
                     max_lo: 100]
    }
}
```

Controlling Version Checking

- GORM creates a version column by default
- Can turn off
- Can switch to a timestamp (from a number)

Turn Off Version Column

```
class Question {
  static hasMany = [answers:Answer]
...
  static mapping = {
    table 'question_authority'
    version false
  }
}
```

Timestamp for Version

```
class Question {
  static hasMany = [answers:Answer]
  Timestamp version
...
  static mapping = {
    table 'question_authority'
  }
}
```

Controlling Naming Globally

org.hibernate.cfg.ImprovedNamingStrategy

- Tables: UserRole => USER_ROLE
- Columns: userName => USER_NAME
- You can customize
  - Organization standards for naming
  - Audit logging
  - etc.

Custom Naming Strategy

```
import org.hibernate.cfg.ImprovedNamingStrategy

class MyNamingStrategy extends ImprovedNamingStrategy {
    // Turn off underscores on column names.
    String propertyToColumnName(String propertyName) {
        return propertyName
    }
}
```

Enable Custom Naming Strategy

DataSource.groovy:

```
hibernate {
    cache.use_second_level_cache = true
...
    naming_strategy = com.opi.MyNamingStrategy
}
```

Exporting the Schema

Can ask GORM to output schema DDL:

```
> grails -Dgrails.env=uat schema-export
```

- Uses Hibernate
- Can specify an environment

Workshop

Implement the following:

- The Question table should be called 'topics'
- Create a join table for Question and Answer. Name it 'topic_responses'
- Add an index on the 'user' table property username
- Create a unique constraint on the 'user' table property username
- Use the dbconsole to verify success

Dealing with Legacy Databases

- Data source configuration
- Composite primary keys
- Reverse engineering a database

Data Source Configuration

In DataSource.groovy

- Can create custom environment configurations for each database and environment
- dbCreate
  - 'create-drop' in development
  - 'update' if schema is already set up in database
  - 'validate' to turn off making any changes - just checks schema
  - null to turn off all options, even validation

Composite Primary Keys

- Most useful for existing databases
- Avoid if you can, Grails works best with single unique ID.
- Class must implement Serializable
- Must implement equals() and hashCode()
- Odd syntax to look up objects

Composite Primary Keys - Example

```
class User implements Serializable {
    String firstName, lastName, password
    String email
    String userName
    static mapping = {
        id composite: ['email', 'userName']
    }

    boolean equals(o) {
        ...
    }

    int hashCode() {
    }
```

Composite Primary Keys - Get

To call `User.get()`, you need a unique ID:

```
User user = User.get(new User(
  email: 'bob@objectpartners.com',
  userName: 'bobmarley'))
println user
```

You are not creating a new user, just finding one.