

Basic GORM

The Grails Domain Layer

Code and Slides

Git Branch: [basicGormStart](#)

Slides: [04 Basic Gorm](#)

In This Session

We'll cover how Grails interacts with your database.

GORM - Grails Object Relational Mapping

- GORM provides the database, or domain, layer for Grails
- Based on Hibernate - really just a thin wrapper
- Use object-oriented syntax to access the database
 - `Employee.list()` returns a list of employees
- Populates domain objects from the database values
- GORM conventions make you more productive

Hibernate

- Helps with the paradigm mismatch between object-oriented software and relational DBs
- Maps between Groovy/Java data types and SQL/JDBC data types
- Extensive support for queries, populating domain objects from query results
- Manages transactions and caching
- Hibernate wraps your domain classes with dynamic proxies
 - Knows if an object has been modified (is dirty)
 - Marshals and unmarshals from JDBC types

grails-app/domain

- Where the GORM magic happens
- GORM wraps each class under this folder
- Each class maps to one table in the database
- Each property on a class maps to a column in the table
- By default, connects to one database (can have more)
- With Grails, all things can be customized

Creating a Domain Class

- Create it under **grails-app/domain**
- Typically in a package underneath this folder
- Can use your favorite text editor, IDE
- ...or the command line

Generated Domain Class

```
package com.opi
```

```
class Question {  
    static constraints = {  
    }  
}
```


Adding some properties

```
package com.opi

class Question {
    String title
    String text
    String answer
    String username

    static constraints = {
    }
}
```

What does this do?

GORM will create a table named QUESTION, with the following columns:

- ID - a unique sequence number
- VERSION - used to prevent concurrent modifications
- TEXT - your property
- TITLE - your property
- ANSWER - your property
- USER_NAME - your property

Some default GORM conventions

- GORM creates the database schema (using Hibernate's schema creation)
- Each domain class maps to a table
- Each property in the domain class maps to a column in the table
- Each table has a unique key field called 'ID'
- Don't worry, you can customize things to use legacy databases
- You can also tell GORM not to create the schema (typical for production)

Let's see it

```
> rails run-app
```

```
...
```

```
| Server running. Browse to http://localhost:8080/questionApp
```

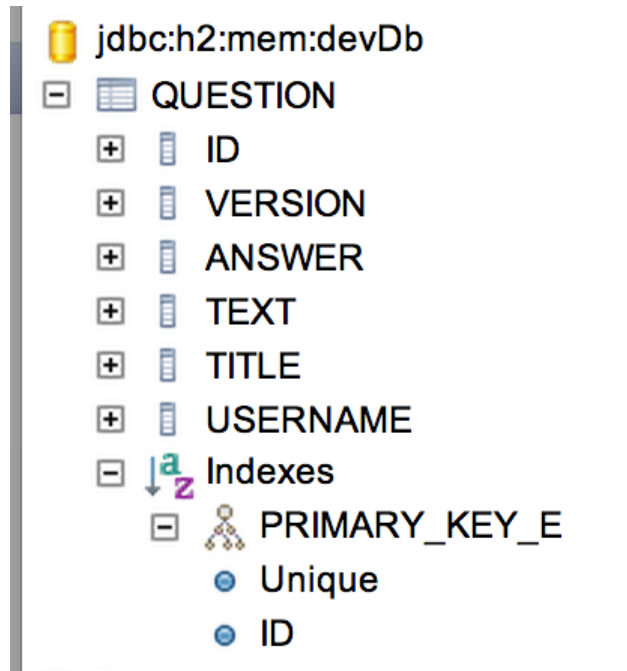
The DB console

- Starting with Grails 2.0
- Can view the in-memory H2 development database.
- Can run queries, updates, and so on.
- Very handy view into your application's database.

<http://localhost:8080/questionApp/dbconsole>

Using the DB console

The QUESTION table looks like:



H2 Built-in Database

Grails includes the H2 in-memory database

- Written in Java
- Automatically starts in development environment
- DB cleared and refreshed each time you run application

Typical Workflow

- Initial development
 - Let GORM and Hibernate manage schema
 - Quickly change database, especially with H2
 - Every developer has a DB
 - No maintenance of DB (restarts each time)
- As things settle down
 - Examine schema for things like performance
 - Modify domain classes as appropriate
 - Test on actual database (Oracle, SQL Server, etc.)
- Into production
 - Control domain class changes
 - Turn off dbCreate mode
 - Manage data migrations

What to do with Domain Classes?

Now that you set up the database, what next?

- Populate and manage data
- Query database
- Validate data

Basic CRUD operations

```
def q = new Question(title:'Help me',  
    text:"I've fallen and I can't get up.")  
q.save()
```

```
def q = Question.get(id)  
q.title = "Question?"  
q.save(flush: true, failOnError: true)  
q.delete()
```

```
def questions = Question.list(offset:10, max:20,  
    sort:"title", order:"asc")
```

GORM CRUD Operations

- GORM augments your domain classes using the Active Record pattern (similar to Rails).
- With this style, you don't need a DAO layer.
- You can save, update and delete from the domain objects.
- You can still use services to control transaction boundaries.
- The list() method automatically supports pagination.

Using Dynamic Finder Methods

Grails also creates dynamic finder methods

- `findBy*` finds one row
- `findAllBy*` finds all rows

```
def questions = Question.findAllByTitle("Shouldnotgethere  
bytecode error?")
```

```
questions = Question.findByDateCreatedBetween(firstDate,  
secondDate)
```

```
questions = Question.findAllByTitleLike('%bytecode%')
```

Validation

GORM automatically validates against the constraints you define:

```
class Question {  
    String title  
    String text  
  
    static constraints = {  
        title nullable: false, blank: false, maxSize: 200  
        text nullable: false, blank: false, maxSize: 10000  
    }  
}
```

Using validation in your code

- `validate()` runs the validation
- GORM will automatically validate on calls to `save()`
- `hasErrors()` returns true if there are errors
- The field *errors* holds the errors
- Uses underlying Spring validation error types for the errors
- By default, all domain class properties are not nullable. (ie. they have an implicit *nullable: false* constraint)

Built-In Constraints Include

blank

creditCard

email

inList

matches

max

maxSize

min

minSize

notEqual

nullable

range

scale

size

unique

url

validator

Checking Validation

```
Question q = new Question()    // No required fields
```

```
if (q.validate()) {
```

```
    // OK ...
```

```
} else {
```

```
    // Has errors...
```

```
}
```

```
if (q.hasErrors()) {
```

```
    // Houston, we have a problem...
```

```
}
```


Validation Messages

- Grails comes with default messages in the **grails-app/i18n** folder
- You almost always want to change the messages

Edit `grails-app/i18n/messages.properties` and add:

```
question.title.blank=You must enter a title.
```

```
question.text.blank=You must enter the question text.
```

Custom Validators

- Grails has a good set of built-in validators
- Sometimes you need more
 - Compare multiple values
 - Perform an external look up
- Create a custom validator

Custom Validator Code

```
class Question {  
    ...  
    static constraints = {  
        title blank: false, maxSize: 200,  
        validator: { val, obj ->  
            if (val.contains('JVM')) {  
                return "jvm.questions.not.allowed"  
            }  
        }  
        text nullable: false, blank: false, maxSize: 1000  
    }  
}
```

Unit testing domain classes

When using the command line to create a domain class, Grails creates a unit test as well.

Spock is the default testing framework.

Generated unit test

```
import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(Question)
class QuestionSpec extends Specification {
    def setup() {
    }
    def cleanup() {
    }
    void "test something"() {
    }
}
```

Adding a test method

```
void "Question should validate"() {  
    when:  
        def q = new Question(title: 'What is def?',  
            text: 'Please explain this thing called def.',  
            answer: "def is an alias of Object",  
            username: "opie")  
  
    then:  
        q.validate()  
}
```

Running Tests

Run this test

```
> grails test-app com.opi.Question
```

Relating Domain Objects

GORM supports:

- one-to-one
- many-to-one
- one-to-many
- many-to-many

Adding More Domain Objects

Create 2 new domain objects

```
> grails create-domain-class com.opi.User
```

```
> grails create-domain-class com.opi.Answer
```

Simple Many to One

```
class Question {  
    User user  
    String username  
    ...  
}
```

```
class Person {  
}
```

Relating Domain Objects - hasMany

```
class Question {  
    static hasMany = [answers: Answer]  
    String answer  
    String title  
    String text  
    ...  
  
    static constraints = {  
        title nullable: false, blank: false, maxSize: 200,  
        ...  
    }  
}
```

Mapping the Other Side

```
class Answer {  
    static belongsTo = [question:Question]  
  
    String text  
  
    static constraints = {  
        text nullable: false, blank: false, maxSize: 10000  
    }  
}
```

But Wait, There's More

There's *a lot more* to relationships:

- GORM II session
- Online docs
 - <http://grails.org/doc/latest/guide/GORM.html>

Conveniences - Modified Dates

```
class Question {  
    static hasMany = [answers: Answer]  
  
    String title  
  
    ...  
    Date dateCreated  
    Date lastUpdated  
    static constraints = {  
        ...  
    }  
}
```

Conveniences - implicit id & version

```
class Question {  
    Long id    // Inserted by GORM, not you  
    Long version  
    static hasMany = [answers: Answer]  
  
    String title  
    ...  
  
}
```

Workshop - Basic GORM - 15 min

- Create a class for a User
 - user name
 - first name
 - last name
 - email address
- Allow a User to be an author of each question and each Answer.
- Choose a reasonable size for the name fields, such as 30 characters.

Bonus:

- Add a custom validator to reject any user with 'Justin' and 'Bieber' as the first and last name.

Workshop Solution

How do you know you are successful?

- Update the unit test to prove a User is necessary to save a Question.

See 'introToGormFinish' Branch

- More details
- One possible solution
- Additional unit tests

For More on GORM

<http://grails.org/doc/latest/guide/GORM.html>

<http://grails.org/doc/latest/ref/Constraints/validator.html>

The GORM II session

Plugin:

- <http://www.grails.org/plugin/database-migration>