

Domain Class & GORM Part I

Sang Shin
JPassion.com
“Learn with Passion!”



Topics

- What is Domain class & GORM?
- CRUD Operations
- Dynamic finders
- Validations
- Error messages
- Custom O/R mapping
- Events
- Automatic timestamping

What is Domain Class & GORM?

What is Domain Class?

- Represents Model (of MVC)
 - > Things to be manipulated
- Domain objects holds state
 - > Through properties
- Domain class has a corresponding table
 - > Domain object has a corresponding row
 - > Properties of Domain class have matching columns

What is GORM?

- GORM (Grails Object Relational Mapping) provides mapping between domain objects and database tables
 - > Accessing, saving, creating, updating operations in your Grails code are performed by GORM
- Under the hood, GORM uses Hibernate by default
 - > Other DB technologies are also supported: MongoDB, CouchDB, Redis, etc
- Leverages dynamic nature of Groovy
 - > Dynamic typing, meta-programming, closure, etc
- Convention of configuration
 - > Example: Name of Domain class is the name of the table

Steps for Creating Domain Class

- Step #1: Create Domain class
grails create-domain-class org.jpassion.Student
- Step #2: Add properties to the Domain class
- Step #3: Add validation constraints
- Step #4: Add relationships to other Domain classes
- Step #5: Add custom ORM features
 - > Custom table/column names, Caching strategy, Locking, Fetching, etc
- Step #6: Add event handling

Creation, Read Update, Delete (CRUD) Operations

CRUD Operations

- Create - Grails transparently adds an implicit *id* property to your domain class which you can use for retrieval:

```
def student =  
  new Student(name: "David",  
              age: 22)  
student.save()
```

- Read (more info. next slide)

```
def s1 = Student.get(1)  
def s2 = Student.read(2)  
def s3 = Student.load(3)
```

- Update

```
def s = Student.get(1)  
s.name = "Bob"  
s.save()
```

- Delete

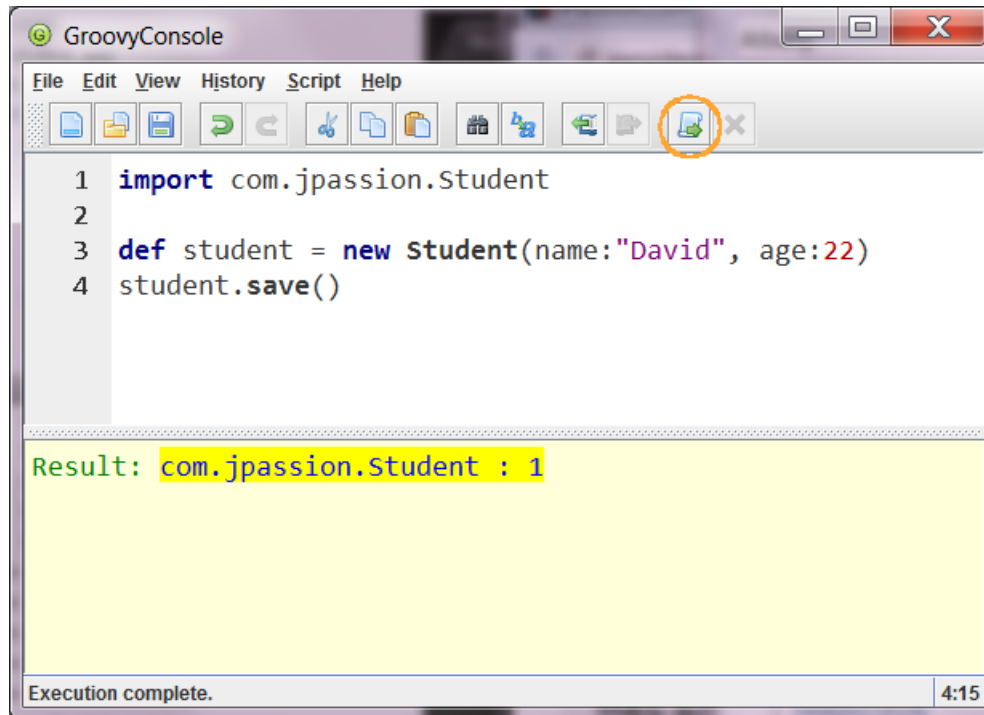
```
def s = Student.get(1)  
s.delete(flush:true)
```


3 Schemes of Read Operation

- `Student.get(1)`
 - > Automatic dirty detection by Hibernate is on as default
 - > Change can be persisted automatically via Hibernate dirty detection (during flush and commit)
- `Student.read(1)`
 - > Similar to the get method except that automatic dirty detection is disabled (during flush and commit)
 - > The change to the instance after `read()` will not be persisted until explicit `save()` is called- use this if you want to make sure you don't accidentally persist unintended change
- `Student.load(1)`
 - > Returns a proxy instance

Grails Console

- Extended version of the Groovy console but with Grails runtime
- Unlike “Rails console”, it does not share the same runtime with running Grails app



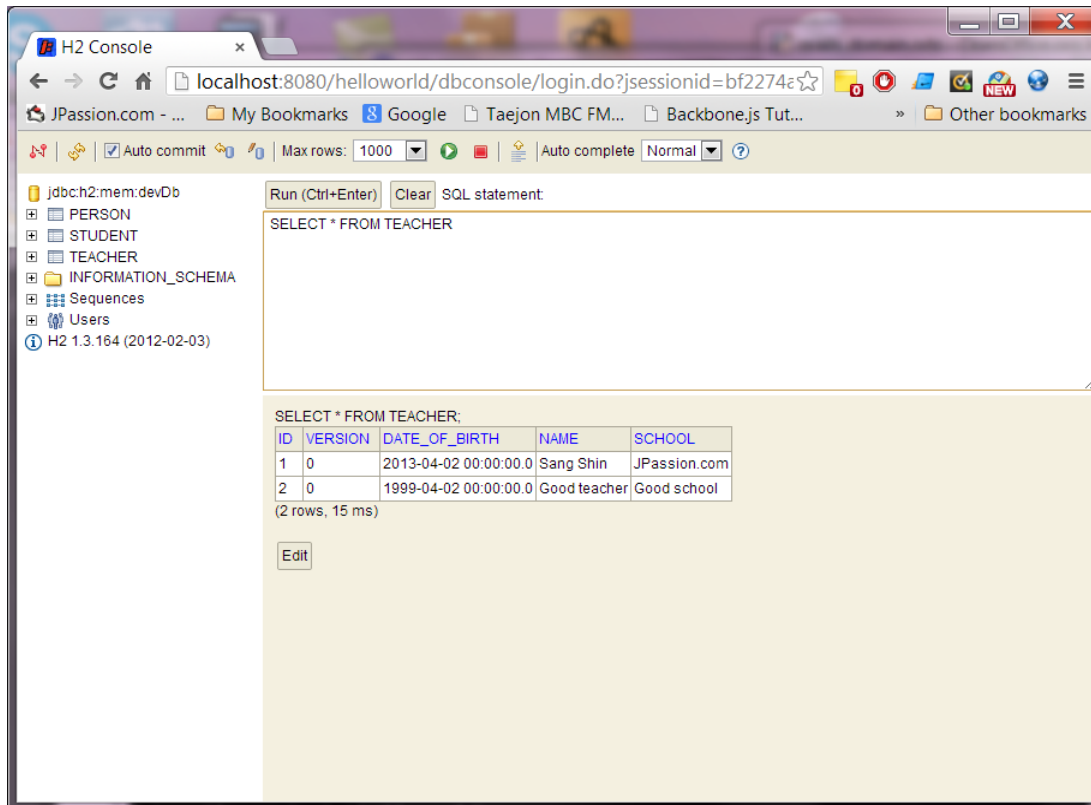
The screenshot shows a window titled "GroovyConsole" with a menu bar (File, Edit, View, History, Script, Help) and a toolbar. The toolbar contains icons for file operations and execution, with the execution icon (a green play button) circled in orange. The main text area contains the following Groovy code:

```
1 import com.jpassion.Student
2
3 def student = new Student(name:"David", age:22)
4 student.save()
```

Below the code, the execution result is displayed on a yellow background: "Result: com.jpassion.Student : 1". At the bottom of the window, a status bar shows "Execution complete." and the time "4:15".

Database Console

- Browser based H2 db manager which comes with Grails
- <http://localhost:8080/<app-context>/dbconsole>



Bootstrapping

- grails-app/conf/BootStrap.groovy
 - > *init(..)* gets executed when Grails starts
 - > *destroy(..)* gets executed when Grails shuts down
- Used for any start-up and clean-up tasks
 - > Test data creation can be done in *init(..)*

Lab:

Exercise 1: CRUD Operations

Exercise 2: Study Scaffolding Code

5626_grails_domain.zip



Dynamic Finders

Getting Domain Instances

- Listing instances using *list(..)* method

```
def books = Book.list()
```

```
def books = Book.list(offset:10, max:20)
```

```
def books = Book.list(sort:"name", order:"asc")
```

- Retrieval by Primary key or keys

```
def book = Book.get(23)
```

```
def books = Book.getAll(23, 93, 81)
```

Dynamic Finders

- A dynamic finder looks like a static method invocation, but the methods themselves don't actually exist in any form at the code level
- They get constructed dynamically during runtime via Groovy's *missingMethod* scheme

Dynamic Finders Examples

- Let's say we have Domain class

```
class Student {  
  String name  
  Date birthday  
  int age  
}
```

- Examples of dynamic finders

```
def student = Student.findByName("Sang Shin")  
student = Student.findAllByNameLike("Sang S%")  
student = Student.findAllByBirthdayBetween( firstDate, secondDate )  
student = Student.findAllByBirthdayGreaterThan( someDate )  
student =  
Student.findByNameLikeOrBirthdayLessThan( "%Something%",  
someDate )
```

Comparators used in Dynamic Finders

- InList - In the list of given values
- LessThan - less than the given value
- LessThanEquals - less than or equal a give value
- GreaterThan - greater than a given value
- GreaterThanEquals - greater than or equal a given value
- Like - Equivalent to a SQL like expression
- Ilike - Similar to a Like, except case insensitive
- NotEqual - Negates equality
- Between - Between two values (requires two arguments)
- IsNotNull - Not a null value (doesn't require an argument)
- IsNull - Is a null value (doesn't require an argument)

Pagination & Sorting

- The same pagination and sorting parameters available on the list method can also be used with dynamic finders by supplying a map as the final parameter

```
def students =  
  Student.findAllByNameLike("Sa%", [max:3,  
                                     offset:2,  
                                     sort:"age",  
                                     order:"desc"])
```

Lab:

Exercise 3: Dynamic Finders 5626_grails_domain.zip



Validations

Declaring Validation Constraints

- Grails provides a unified and convenient way to define validation "constraints" in the domain class

```
class User {  
    String login  
    String password  
    String email  
    Integer age  
    static constraints = {  
        login(size:5..15, unique:true)  
        password(size:5..15)  
        email(email:true, blank:false)  
        age(min:18, nullable:false)  
    }  
}
```

Built-in Constraints

- *blank:true, nullable:true*
- *creditCard:true*
- *display:false* (Hides the field in *create.gsp* and *edit.gsp*)
- *email:true*
- *password:true*
- *inList:["foo", "bar", "x"]*
- *matches:"[a-zA-Z]+"*
- *min:0, max:100*
- *size:1..10*
- *range:0..10*
- *scale:5*
- *unique:true*
- *url:true*
- *notEqual:"Foo"*
- *validator: {return(it%2) == 0}*

Validation Basics

- You can call the *validate* method on any instance anytime
- Every domain object also has *errors* property
 - > Provides methods to navigate the validation errors and also retrieve the original values.

```
def student = new User(params)
if(student.validate()) {
  // do something with student
}
else {
  student.errors.allErrors.each {
    println it
  }
}
```


Two Validation Phases

- Within Grails there are essentially 2 phases of validation
 - > The first phase: data binding phase (type conversion phase)
 - > The second phase: domain constraints checking phase

Data Binding Phase Validation

- The first phase is data binding which occurs when you bind request parameters onto a domain instance
- Type conversion error is detected in this phase

```
// Type conversion occurs  
def student = new Student(params)
```

```
// You can detect type conversion error  
if(student.hasErrors()) {  
    if(student.errors.hasFieldErrors("login")) {  
        println student.errors.getFieldError("login").rejectedValue  
    }  
}
```

Domain Constraints Checking Phase

- The second phase of validation happens when you call *validate* or *save* method. This is when Grails will validate the bound values against the constraints you defined.

```
if(student.save()) {  
    return student  
}  
else {  
    student.errors.allErrors.each {  
        println it  
    }  
}
```

Lab:

Exercise 4: Validation
5626_grails_domain.zip



Error Messages

Constraints & Message Codes

- The codes themselves are dictated by a convention
 - > If a constraint was violated, Grails will, by convention, look for a message code in the form
 - > [Class Name].[Property Name].[Constraint Code]
- In the case of the blank constraint this would be *user.login.blank* so you would need a message such as the following in your `grails-app/i18n/messages.properties` file:
user.login.blank=Your login name must be specified!
user.login.unique=Fool! value [{2}] must be unique

Lab:

Exercise 5: Custom Messaging 5626_grails_domain.zip



Custom O/R Mapping

Custom O/R Mapping

- Table and column names
- Caching strategy
- Inheritance strategy
- Custom database identity
- Composite primary keys
- Database indices
- Optimistic locking and versioning
- Eager and Lazy fetching
- Custom cascade behavior

We will cover most of these in “Association” presentation.

Custom Table and Column Names

```
class Person {  
    String firstName  
    static mapping = {  
        table 'people'  
        firstName column:'First_Name'  
    }  
}
```

Lab:

Exercise 7: Custom O/R Mapping 5626_grails_domain.zip



Events

Events

- beforeInsert
- beforeUpdate
- beforeDelete
- beforeValidate
- afterInsert
- afterUpdate
- afterDelete
- onLoad - Executed when an object is loaded from the database

beforeInsert & beforeUpdate Event

```
class Person {  
  
    Date dateCreated  
    Date lastUpdated  
  
    def beforeInsert() {  
        dateCreated = new Date()  
    }  
    def beforeUpdate() {  
        lastUpdated = new Date()  
    }  
}
```

Lab:

Exercise 8: Events

[5626_grails_domain.zip](#)



Automatic Timestamping

Automatic Timestamping

- By merely defining a `lastUpdated` and `dateCreated` property these will be automatically updated for you by GORM.

```
class Person {  
  
    Date dateCreated  
    Date lastUpdated  
  
    // The following code is not needed because GORM  
    // does automatic timestamping  
    //def beforeInsert() {  
    //    dateCreated = new Date()  
    //}  
    //def beforeUpdate() {  
    //    lastUpdated = new Date()  
    //}  
}
```

Learn with Passion!
JPassion.com

