Grails – Domain - II

Agenda

- ***Domain Mappings***
- ***Custom Domain Mapping***
- ***Locking Strategies***
- ***Fetching Strategies***
- ***Custom Validations***
- ***GORM DSL***
- ***Questions***

Domain Mapping

Grails by default maps domain class name to table name and domain fields to table columns.

Grails	**Database**
Person	person
PersonAddress	person_address
email	email
firstName	first_name

Custom Mapping

- It is generally done to map domain classes with already defined/existing database

- Defined as the static mapping block within a domain class

```
static mapping = {}
```

Mapping for table names

- Domain say ‘Person’ can be mapped to table ‘people’ instead of ‘person’ as :

```
class Person {
    static mapping = {
        table ‘people’
     }
}
```

Mapping column names

- Domain property, firstName can be mapped to ‘’fname” instead of “first_name” as :

```
class Person {
    String firstName

    static mapping = {
        firstName column: ‘fname’
    }
}
```

Mapping column type

- Domain property of type String can be mapped to any other database data type, say text as :

```
class Person {
     String description

     static mapping = {
          description type: 'text'
     }
}
```

Mapping id field

- We can set auto generated property ‘id’ to some other name, say ‘personId’:

```
class Person {
    String description
     Long personId

    static mapping = {
        id name: ‘personId’
    }
}
```

Mapping composite keys

We can define composite primary keys

```
class Person {
        String firstName
        String lastName

         static mapping = {
         id composite: [‘firstName’,’lastName’]
         }
}
```

Mapping autotimestamp variables

- Properties dateCreated & lastUpdated if exists, gets automatically updated by grails when object is created and updated.
- These updates can be disabled

```
class Person {
        Date dateCreated
        Date lastUpdated

        static mapping = {
        autoTimeStamp:false
        }
}
```

Mapping for sort

- By default all the data fetched from GORM queries is sorted on insertion order
- Default sorting column and order can be changed

```
class Person {
    String firstName


    static mapping = {
    sort "firstName"
    or
    sort firstName:'desc'
    }
}
```

Mapping for tablePerHierarchy

- By default grails create a single class for all the domain classes in inheritance
- In this case table has all the attributes of all the child as well as the parent and it has one additional field 'class' to differentiate among the instances.
- If you want different tables for all the domain classes, you need to set tablePerHierarchy as false

```
static mapping = {
      tablePerHierarchy:false
}
```

Mapping for cascade

- It is used to ensure that association or child objects are also deleted when its parent objects are deleted.
- It is used in case where you don’t have “belongsTo” relation specifies.

```
static mapping = {
      addresses cascade :’delete-all-orphan’
}
```

Mapping for version field

- Used to disable optimistic locking
- Setting it as false will not create any field in database

```
static mapping = {
     version:false
}
```

Demo

- Demo on Custom Mappings
- Demo on table per hierarchy

Locking Strategies

- **Pessimistic Locking** : resource is locked from the time when it is first accessed in a transaction until the transaction is finished, making it inaccessible to other transactions during this time.
- **Optimistic Locking** : resource is not locked when it is first accessed in a transaction, instead state of the resource is saved. Other transactions are able to concurrently access the resource & the possibility of conflicting changes is possible.

By default GORM classes are configured for optimistic locking. It involves storing a version field that is incremented after each update. when update is performed, Hibernate will automatically check the version property against version field in the database and if they differ will throw **StaleStateException**.

Fetching Strategies

- **Lazy Fetching** : By default all the associations are loaded lazily, relationships of an instance will be fetched from database only when it is needed, this is also called lazy initialization

- **Eager Fetching** : It is opposite of lazy, eager will fetch all the associations when object is retrieved.

Sample Classes

```
class Airport
{
String name
static hasMany = [flights:Flight]
}
```

```
class Location {

String city
String country
}
```

```
class Flight{
String number
Location destination
static belongsTo = [airport:Airport]
}
```

```
Airport airport = Airport.findByName(“Gaitwick”)

airport.flights.each{Flight flight->
   println flight.destination.city
 }
```

Lazy Fetching

- Number of queries :
- N + 2 queries
- First query to fetch airport instance, then to fetch flights of airport and 1 extra query for each iteration over flights association to get instance of destination.

Eager Fetching

- Number of queries :
- 2 queries

```
class Airport                              class Flight { destination
fetch:'join' }
{
       String name

      static hasMany = [flights:Flight]

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

Eager Fetching

- With above query flights will be loaded with the airport itself but with 2 different queries
- To load in the same query we could have used

```
static mapping = {
  flights fetch:'join'
}
```

- But it is recommended to use fetch join with only single ended associations and lazy:false with one to many cases

Demo

- Demo on Optimistic locking
- Demo on fetching strategies

Custom Validations

- Custom validators are used to define arbitrary validations, eg User cannot have password same as firstName

```
class Person {
    String firstName
    String password

    static constraints = {
        password(validator:{val,obj->
        if(val.equals(obj.firstName)){ return false }
        })
    }
}
```

Custom Validators

- Validator closure can take upto 3 arguments, first is the value, second is the domain class instance and the third is the Error's object.
- The validator closure can return
  -> null or true to indicate value is valid
  -> false to indicate an invalid value and use the default message code, from message.properties
  -> A string to indicate the error code

GORM DSL(get)

- Retrieves an instance of the domain class for the specified id otherwise null

```
Person.get(1L)
```

- Any change made in the instance retrieved by get will be persisted even without calling save explicitly.

```
person.firstName = “Test” // will update without calling save
```

- Retrieves instance from cache if available otherwise from database

GORM DSL(read)

- Retrieves an instance of the domain class in a read only state for the specified id otherwise null

```
Person.read(1L)
```

- Any change made in the instance retrieved by read will not be persisted until save is called.

```
person.firstName = “Test” // will not be updated without save
```

- Retrieves instance from cache if available otherwise from database

GORM DSL(load)

- Returns proxy for the instance that is initialised on demand.

  Person.load(1L)

- No database call is made until you access any property other than id

- If object is available in cache then object is returned instead of proxy

- If wrong id is provided no exception is thrown till property other than id is referenced.

Demo

- Demo on custom Validations
- Demo on fetching strategies

Questions??

Thank you!!