**What is Grails?**

1. Grails started in 2007 based on
2. But 3.0 based on springboot.
3. Open source, full stack web framework for JVM.
4. “Convention over configuration” and uses Groovy. It wants you to write things in the way they are supposed to be written. Saves from lots of other issues.

**Why Grails?**

1. High productivity on Enterprise Edition as compared to other frameworks of JAVA like Spring MVC or hibernate.
2. Easy to use.
3. Reuse existing java techs under single interface.
4. No XML configurations.
5. More agile dev cycle.
6. Controller built on Spring MVC which saves us from any infrastructural issues. In case we need to switch our ORM or service.
7. Ready to use environment.
8. Easy to use ORM on hibernate i.e. object to database is easy and reusable irrespective of factors like size or type of databases.
9. Build system based on Gradle.
10. Transactional service based on Spring’s transaction abstraction.
11. Availability of additional functionality through use of plugins. The objects of libraries are readily configured as your default configurations.
12. Includes the deployment container, database, build system and test harness.

Build.gradle is the file where we can place our dependencies.

Domain class is the class that has corresponding to database table.

Read PPT

**GRAILS-DOMAIN**

Part of Grails “Model”

Relational Objects: Objects which map with the database tables and operation on that table is done throught these.

ORM: Mechanism through which an object is mapped with database tables is ORM

Domain is gateway from our application interacts with the database. Grails uses Hibernate to map domain classes with databse.

Static constraints : the field level constraints applied to a domain class which can be used to validate.

Static mapping: giving aliases to field names of databases.

Application.yml has precedence over Application.groovy configuration file.

Bootstrap.groovy is executed before application comes UP

Two fields in Domain: Persistent and transient fields.

Persistent fields: All the fields that are mapped to the database. Each will by default are required.

Transient fields: Fields which we may not want to store in DB. But used in applications.

Static transients = [‘name’]

name will make transient field.

We can make explicit setter and getter.

dateCreated automaticall checks for these values

lastUpdated

.validate() function checks for field level validation and returns boolean.

.save() will return the instance itself if save completed else returns NULL instance and has three different properties or arguments

flush – flush the objects in hibernate session to DB (true/false)

failOnError – application to terminate or not (true/false)

validate – to validate or not (true/false)

**Custom Mapping**

Map domain classes with already existing database.

Static mapping = {

table ‘people’

firstname column:’fname’

description type:’text’

id name:’personId’

id composite:[‘’,’’]

autoTimeStamp:false

sort ‘firstName’

or sort firstName:’desc’

tablePerHierarchy:false in the parent table

subclass cascade: ‘delete-all-orphan’

version:false

}

**Locking**

* Pessimistic Locking: from time to time and once accessed by one then locked for others.
* Optimistic Locking: by default, resource is not locked but state is saved with the help of version field and if it differs from last transaction then throws StaleStateException.

**Fetching**

* Lazy: by default, relationship will be fetched only when needed with that instance.
* Eager: all associations when object is retrieved.

With one to many class lazy:false

with one to one property.fetch:’join’

**Custom Validations**

password(validator:{val.obj->if(val.equals(obj.firstName)){return false}

null//true

false/message code

string error code

**GORM DSL**

Retrieves an instance of domain by ID.

Class.get(1L)

Class.firstName=””

Always saves the transaction to database without even save().

Class.read(1L)

Class.firstName=””

Does not saves the transaction to database until save().

Class.load(1L)

returns proxy for instance.

**GORM**

.findBy(val)

.findAllBy(val)

.list() select \*

.list(offset: val1, max:val2)

.list(sort:val,order:asc/desc)

.get(only id value)

.getAll(all id values)

.getAll(i..j range of id values)

.collection.find{colsure}

.removeFromCollection(object)

**Dynamic Finders**

.findByPropertyLike(“regex”)

.findByPropertyBetween(first,second)

.findByPropertyLikeOrBetween(

.findOrCreateByProperty(value)

.findOrSaveByProperty(values)

.countByProperty(value)

All like findBy

**Criteria**

.withCriteria{colsure}

.withCreateCriteria.list{

and/or/not{

comparator(property,values)

}

}

.get{}

.listDistinct{}

.count{}

**projections** is used for what columns to output

projections{

property(fieldname)

distinct(“”)

avg(“”)

max(“”)

sum(“”)

min(“”)

rowCount()

count(“”)

countDistinct(“”)

groupProperty(“”)

}

**CONTROLLERS**

HTTP request comes directly to Controller which then initializes the required Model and selects the appropriate View.

Grails create-controller sample.

A new instance is created for evry HTTP request.

**Actions** are groovy closures and they map to URI.

Def myFirstAction(){

render “Sample controller Action”

}

Actions name starting with “get\*\*” shouble be avoided.

**Scaffolding** allows to use dynamically genereated code to get you started.

Def scaffold = true

def scaffold = Domain\_class

* Dynamic Scaffolding :- It will provide domain class on runtime.
* Static Scaffolding :- From terminal generate all.

**Default Actions**

* def index(){ } It automatically first looks in for index action.
* Static defaultAction = “Action name”

By default all methods can act on every actions. In case we want to restrict methods over actions....

Static allowMethods = [action1:’POST’,action2:[’POST’,’GET’,’DELETE’]]

**Creating a Model**

def show={

render(view:’show’,model:[song:Song.get(1)]  
}

**model has all the GORM queries written**

forward(controller : ‘’,action : ‘’)

redirect(controller : ‘’,action : ‘’,params:[],url:’’)

forward and redirect are different as redirect changes to URL to the controller action where it is redirected.

**Controller Object Scopes**

servletContext:- accross the application

session:- for current user

request:- for current request#

params:- map of parameters passed from view to controller\*

\* not really a scope

flash:- current request and next request used mostly in case of redirect.

#

ActionName

actionUri

controllerName

controllerUri

response

**Controller Scopes**

Prototype: one object of controller per action/request

Session: one object of controller per session

Singleton: one object of controller per application

Before Interceptor : procession before every action

def beforeInterceptor = [action:this.&myAction]

After Interceptor : processes after ecery action

def afterInterceptor = [action:this.&myAction]

**Data Binding or Updating data into database**

* Using contructor, by creating new object of domain class and then .save(flush:true)

and redirect(action:’index’)

* Book.properties = params //updating record

and redirect(action:’index’)

* bindData(dest, src, include:[], exclude:[]) class,metaClass have to be excluded always

In cases when we need pass everything then we don’t pass the include parameter

**VIEWS and TAGLIBS**

* Views are outputs of our requests.
* GSP is Groovy + HTML for rendering views in Grails.
* It can have static and dynamic content.
* Values in the scopes are rendered in the gsp.
* Views are rendered to client as HTML.

**TagLibs**

All tags are started with “g:”

<g:form name = “” action= “” id=””></g:form>

and So on..................

**Custom Taglibs**

The logic of what we want to render ton the gsp can be defined by user and those custom tags are then used at GSP. These are the customizable tags where we tell the gsp what a tag does.