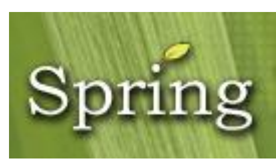
GET TO KNOW GRAILS

UNDERLYING TECHNOLOGIES



CONDITIONAL ENVIRONMENTS

- Environment conditionals allow for variations for a specific environment
  - Database
  - Initialization options
  - More
- Three environment choices
  - Development
  - Testing
  - Production

SCRIPT-BASED CODE GENERATION

- grails create-app [app name]
- grails create-controller [controller name]
- grails create-domain-class [class name]
- grails create-service [service name]
- grails create-unit-test [test name]
- grails create-tag-lib [taglib name]
- grails generate-all [class name]
- grails generate-views [class name]

BUILDING, DEPLOYING AND MANAGING

- grails clean
- grails compile
- grails console
- grails doc
- grails install-plugin
- grails run-app
- grails war

CREATE WEB APPLICATION

- Create a new directory which should contain your Grails application and type in the following command
  - grails create-app guestbook

- This command creates the directory structure and the basis configuration of your new web application

- This created already a full web application

DIRECTORY STRUCTURE

SCAFFOLDING

- Grails supports a feature known as ‘scaffolding’ which involves the generation of a CRUD (Create/Read/Update/Delete) interface for a given domain class
  - Generate-all command allows you to generate an implementation of CRUD including a controller and views for the given domain class
  - grails generate-all [domain class name]

IMPORTANT FILES

- conf/DataSource.groovy
  - Database connections
- conf/UrlMapping.groovy
  - Routing
- conf/BootStrap.groovy
  - Bootstrap file
- conf/Config.groovy
  - Configurations (MIME mappings, more…)
- Grails supports the concept of per environment configuration. Both the Config.groovy file and the DataSource.groovy file within the grails-app/conf directory can take advantage of per environment configuration

DATABASES

- Db configuration is in conf/DataSource.groovy
- Can define at environment level or global level
- Defaults to HSQL
- Change to Oracle by:
  - Dropping ojdbc6.jar in the lib folder
  - Changing DataSource.groovy to point at Oracle

DATABASE CREATE SCHEME

- dbCreate
  - create-drop: creates tables on startup and drops them on shutdown (DEV)
  - create: creates tables on startup, deletes data on shutdown (DEV, TEST)
  - update – creates tables on startup, saves data between restarts (TEST, PROD)Db configuration is in conf/DataSource.groovy

DATABASES

- conf/DataSource.groovy

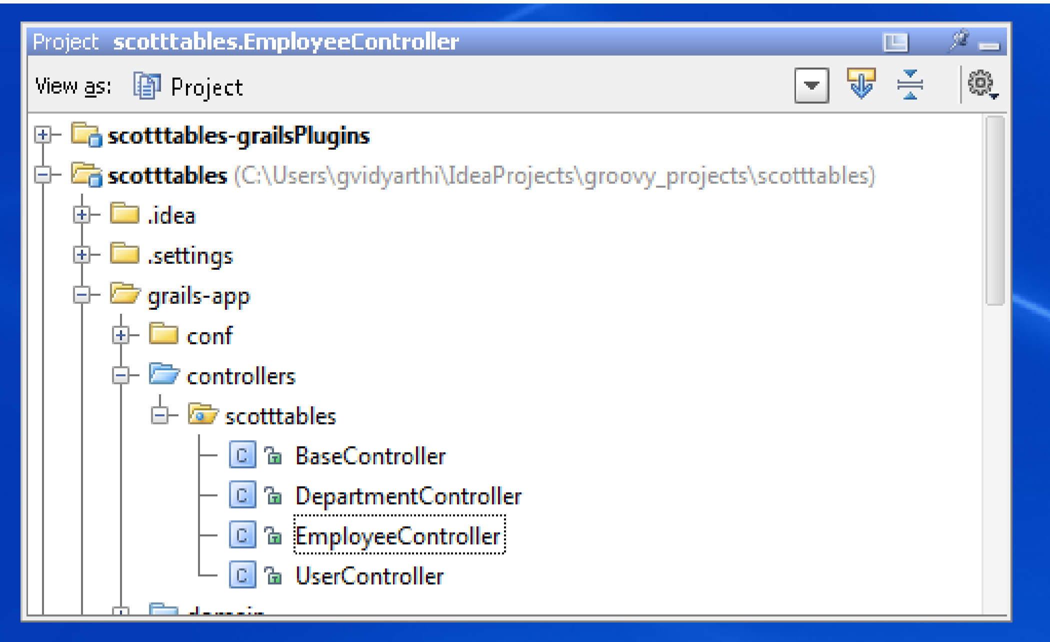
```
import org.hibernate.dialect.Dialect

dataSource {
   pooled = true
   driverClassName = "oracle.jdbc.driver.OracleDriver"
   username = "scott"
   password = "tiger"
   dialect = "org.hibernate.dialect.Oracle10gDialect"
}
hibernate {
   cache.use_second_level_cache = true
   cache.use_query_cache = true
   cache.provider_class =
'net.sf.ehcache.hibernate.EhCacheProvider'
}
// environment specific settings
environments {
   development {
      dataSource {
         dbCreate = 'update'
         url = "jdbc:oracle:thin:@localhost:1521:orcl1"
      }
}
```

```
  test {
      dataSource {
         dbCreate = "update"
         url = "jdbc:h2:mem:testDb"
      }
   }
   production {
      dataSource {
         dbCreate = "update"
         url = "jdbc:h2:prodDb"
      }
   }
}
```

ROUTING

- Route rules found in conf/UrlMapping.groovy
  - Default route:
    - application/controller/action/id[?params]
    - http://localhost:8080/scotttables/employee/create



ROUTING

- Additional rules and restrictions can be applied here

```
static mappings = {
"/product/$id?"(controller:"product"){
                action = [GET:"show", PUT:"update",
                DELETE:"delete", POST:"save"]
        }
}
```

GRAILS MVC

MVC FRAMEWORK

- Standard MVC implementation

- Sits on top of Spring MVC
  - Reduces repetition of XML developers must maintain
  - Provides DSL for spring configuration

VIEWS

- Groovy Server Pages
  - GSP extension
  - Based on JSP pages
  - Use
    - Tag libraries
    - Expressions
    - similar to JSP EL but allows any expression within ${…}

TAG LIBRARIES

- Ordering and sorting

```
<g:sortableColumn property="releaseDate"
        defaultOrder="desc"
        title="Release Date"
        titleKey="book.releaseDate" />
```

- Form display

```
<g:form name="myForm"
        action="myaction" id="1">
        <g:passwordField name="myPasswordField“
                         value="${myPassword}" />
        ...
</g:form>
```

- Formatting

```
<g:formatDate format="yyyy-MM-dd" date="${date}"/>
```

INTERNATIONALIZATION SUPPORT

- Based on Java i18n specifications
- Define properties files with language specific entries
- Views can read i18n entries with a tag:

```
<g:message code=‘my.message’ args=‘${[‘One’, ‘Two’]}’/>
```

- Tag libraries can read entries with code:

```
g.message(code: ‘my.message’, args: [‘One’, ‘Two’])
```

MODELS

POGOs

- Fields default to private
- Getters and setters provided automatically
- Must use wrapper objects – no primitives (1.5?)
- Field validation defined within:
  static constraints = { date(nullable: false) }
- Automatic parameter mapping from view
  employee.properties = params

GORMS

- Grails Object Relational Mapping (uses Hibernate3)

- Querying with GORM
  - Employee.list()
  - Employee.list(offset:10, max:20)
  - Employee.list(sort:'ename', order:'asc')
  - Employee.get(400)

- Dynamic methods for finding
  - Employee.findAllByJob(" MANAGER")
  - Employee.findAllByManager(params.managerid)
  - User.findByUseridAndPassword(params.userid,params.password)

- Relational mapping defined within
  - One to many: (In Department domain class)
    static hasMany = [employees: Employee]

GORMS

- One to one: (in Employee domain class)
  Department department

- Cascading updates and deletes
  - static belongsTo =[department:Department]
  - decides which class owns the relationship and whether updates and deletes should be cascaded from the owning class to the its childs.

- Eager and lazy fetching
  - Associations are by default lazy
  - We can specify eager mode using static fetchMode
    - In Department domain class:
      - static hasMany = [employees: Employee]
      - static fetchMode =[employees:" eager"]

GORM WITH LEGACY DATABASE

- Map the table to an existing table(most of the legacy table naming convention will not be same as what hibernate expect)

- Turn off the optimistic locking so hibernate will not look for the version column on the table
  - E.g. In the Employee.groovy class:

```
static mapping = {
                table("EMP")
                version false
                 columns {
                    id (column:'EMPNO', generator: 'sequence',  params:[name:
'EMP_SEQUENCE'])

}
```

GORM CACHING

- Provides the same caching mechanisms that can be found with Hibernate3

- Updates to DataSource.groovy can toggle caching:

```
hibernate {
    cache.use_second_level_cache = true
    cache.use_query_cache = true
    cache.provider_class =
                    'org.hibernate.cache.EhCacheProvider'
```

CONTROLLERS

- House actions (methods) that respond to a request
- Actions should map to GSP of same name
- Action is not mandatory in the URL. Grails execute the default action.
  - index is the default action
  - If only one action, it becomes the default action.
  - def defaultAction =‘list’

CONTROLLERS

- List of attributes automatically injected into Grails:

Attribute	Description
actionName	The name of the currently executing action
actionUri	The relative URI of the executing action
controllerName	The name of the currently executing controller
controllerUri	The URI of the executing controller
flash	The object for working with flash scope
log	An instance of the org.apache.commons.logging.Log
params	A map of request parameters
Request, response, session, servletContext	

- Data Binding

```
def employeeInstance = new Employee()
employeeInstance.properties = params  or
bindData(employeeInstance, params,[exclude/include: name of properties])
Or  def employeeInstance = new Employee(params)
```

ACTION RESPONSES

- All actions respond in one of three ways:
  - redirect
    - Equivalent to response.sendRedirect
    - Can specify actions, controller, params, and more
    - redirect (controller: 'user', action: 'login')

- return
  - Returns a value and calls a GSP of the same name (action method 'list' will forward to list.gsp)

- render
  - Calls a GSP by name
  - ability to pass arguments
  - render(view: "create", model: [userInstance: userInstance])

SERVICE

- Service layer to encapsulate business logics
  - Created using grails target : create-service
  - Scope of Services: prototype, request, flash, flow, conversation, session, singleton(default)
  - Scope and be changed using : static scope = ‘request’
  - Services can be injected by name into a controller. Grails support dependency injection by convention.
    - Eg in EmployeeController.groovy
    - def employeeService // injects an instance of EmployeeService.groovy
  - All services have transaction demarcation enabled by default- to disable : static transactional = false

```
class EmployeeService{
        static transactional = false
        static scope = ‘request’


}
```

TRANSACTION

- Grails, since it is built on top of Spring and Hibernate, uses their underlying mechanism to deal with transactions.

  - There are two ways to do transactions in Grails:
    1. Method-level Transaction: By default, each Service class in grails has a transactional property set to true. So if any public method in a Service class throws a RuntimeException or any Error, the transaction will be rolled back. This default transaction property can be turned off by setting transactional to false.
    2. Block-level Transaction: The first approach treats everything inside the method as a single transaction. But sometimes we need more control than that.
       - static *withTransaction* method of the Grails Domain object. It doesn't matter which Domain object you call the *withTransaction* method on. It is just to mark the start and end of a transaction.
       - With this approach we don't have to be inside a service class to use a transaction.