

Grails Application Development

Part 4 – Controllers



Objectives

- To understand How controllers work, the generated code and the relationship between actions and pages



Session Plan

- What is a Controller?
- Understand the URL Structure (REST)
- Generate Controllers
- Walkthrough the code
 - 3 Rs
 - Scopes
 - message function
- Respond with No views



What is a Controller?

- Handles the request that comes from the browser
- Executes the logic
- And sends a response back (mostly HTML)



More technically

- Receives the HTTP request from a client
- Parses the request data
- Executes the logic - does Database operations
 - Chooses an appropriate response view
 - Sets the data as model
 - Sends the view back to the browser
- Or
 - Occasionally redirects the browser to a different controller action/view
- Or
 - Sends the data in XML / JSON format (WebServices)



URL Structure & REST

- So far we have created a controller and set a scaffold
- Here all Controller action code and views are generated dynamically by Grails
- We can run the app and focus on one entity
- Study URL patterns
- URLs of Grails application follows REST
- REST stands for Representational State Transfer



URL Structure

- When we click on the **circle** controller link in the main page browser displays the list of circles with the URL
`http://localhost:8080/LearnAround/circle/list`
- When we click an item in the list it shows the record with
`http://localhost:8080/LearnAround/circle/show/1`
- When we click “edit” button
`http://localhost:8080/LearnAround/circle/edit/1`
- When we click “New Circle” button
`http://localhost:8080/LearnAround/circle/create`
- After edit/create controller goes back to Show screen
- After delete we get the list screen back



URL Structure

- in General the URL is of the form

`http://<Server:Port>/context-root/controller/action/id`

- CircleController class name becomes “circle”
- id - represents the id field of the record and applicable for actions - Show & Edit
- id is not needed for List & Create actions



Grails Commands

- Create a new empty controller (name without controller)

```
grails create-controller <package>.Circle
```

- This creates a controller class CircleController.groovy with an empty index action
- Create a controller with code for MVC

```
grails generate-controller <package>.Circle
```

- This creates code for complete CRUD actions
- Wont work without generating views



Controllers

- Single instance shared by each request (Instance per request before Grails 2.2)
- Configurable by `grails.controllers.defaultScope` setting



The Controller class

- How a controller looks?

```
class CircleController {  
  
    static allowedMethods = [save: "POST",  
                             update: "PUT", delete: "DELETE"]  
  
    def index(Integer max) {  
        params.max = Math.min(max ?: 10, 100)  
        [circleInstanceList: Circle.list(params),  
         circleInstanceTotal: Circle.count()]  
    }  
    //other functions/actions...  
}
```

- allowedMethods is a map - which HTTP methods should be used for certain action



- Actions are implemented as functions

```
def list() {  
    //Code for List action  
}
```

- Earlier versions of rails this was done using closures

```
def list = {  
    //Code for List action  
}
```



- Return - Model data from controller action (like index)
 - Grails will choose a view matching the action name (index.gsp)
- Render - Method to specify a non-default view (different than action name)

```
render (view: 'show', model: circleObj)
```

 - Output will be show.gsp with circleObj as model data
- Redirect - to another URL (controller and action)

```
redirect(action: 'list', controller: 'circle')
```

 - Grails will choose a view matching the action name (index.gsp)



Respond without Views - Text/Html

- render can be used for sending text to client (browser)
- Add a new action in CircleController

```
def name(Long id) {  
    def circleInstance = Circle.get(id)  
    render "${circleInstance.name}"  
}
```

- When you access the url

`http://localhost:8080/LearnAround/circle/name/1`

- You can see the name of the Circle in the browser
- You can also send html text like this

```
render "<h1>${circleInstance.name}</h1>"
```



- We add another action

```
def xml(Long id) {  
    def circleInstance = Circle.get(id)  
    render circleInstance as XML  
}
```

- when we invoke this with URL

`http://localhost:8080/LearnAround/circle/xml/1`

- We get the following output

```
<circle id="1">  
    <description>describe test circle 1</description>  
    <discussions>  
        <discussion id="1"/>  
    </discussions>  
    <memberships/>  
    <name>Test circle 1</name>  
    <owner id="1"/>  
</circle>
```



- We add another action

```
def json(Long id) {  
  def circleInstance = Circle.get(id)  
  render circleInstance as JSON  
}
```

- when we invoke this with URL

`http://localhost:8080/LearnAround/circle/json/1`

- We get the following output

```
{  
  "class": "com.aradhika.learn.Circle",  
  "id": 1,  
  "description": "describe test circle 1",  
  "discussions": [{"class": "Discussion", "id": 1}],  
  "memberships": [],  
  "name": "Test circle 1",  
  "owner": {"class": "User", "id": 1}  
}
```



Respond without Views - XML & JSON

- Sending response data as XML or JSON
- Service Oriented Architecture
 - Web services
 - RESTful services
- Serve mobile apps with data
- Mobile apps built using native SDK or HTML5 tools can consume this



The new Respond method

- Respond method works on request content type send XML/JSON or view

```
def list() {  
    respond Circle.list()  
}
```

- Sends the view if the url requested is /circle/list
- Sends XML for /circle/list.xml
- Sends JSON for /circle/list.json
- This is used in all generated controller actions
- We will be very specific with JSON services and use render object as JSON



Respond without Views - response object

- Servlet SDKs response object

```
response.getOutputStream << "Topic : ${circleInstance.name}"
```

- When we have a property of type byte[] - array of bytes
 - To store media files like photos of users
 - We could send photo bytes to the client using

```
byte[] photo = userInstance.photo  
response.getOutputStream << photo
```



URL Mapping

- Check urlMappings.groovy in conf folder
- The default contents

```
"/$controller/$action?/$id?(.$format)?"{  
    constraints {  
        // apply constraints here  
    }  
}  
  
"/" (view: "/index")  
"500" (view: "/error")
```



URL mapping

- How to map custom REST urls?
- Parts of URL mapping
 - URL
 - controller , action
 - HTTP method
- Simple

```
"/circles" (controller:"Circle", action: "list",  
                                         method: "GET")
```
- With path variables (id will be part of params or parameter to the action method)

```
"/circle/$id/discussions" (controller: "Circle",  
                           action: "discussions", method: "GET")
```



- One URL many HTTP methods

```
"/circle/$id" {  
    controller="Circle"  
    action = [GET:"get", PUT:"update", DELETE:"delete"]  
}
```



Controller - REST / JSON Samples

```
class CircleController {  
    static allowedMethods = [save: "POST", update: "PUT"  
        , delete: "DELETE"]  
  
    def list() {  
        render Circle.list() as JSON //status set to 200  
    }  
  
    def get(Long id) { //params.id passed as parameter  
        def circleInstance = Circle.get(id)  
        response.status = 200 //set status explicitly  
        render circleInstance as JSON  
    }  
    //Other actions  
}
```



```
def save() {  
  def circleInstance = new Circle()  
  circleInstance.properties = request.JSON  
  def responseBody = [:]  
  if(circleInstance.save(flush:true)) {  
    responseBody.id = circleInstance.id  
    responseBody.message = 'Circle created successfully'  
    response.status = 200  
  }  
  else {    //some validation error  
    responseBody.message = 'Some error(s) exist'  
    response.status = 400  
  }  
  render responseBody as JSON  
}
```



Scopes used in Controllers

- The data used by Controllers are critical to the application
- The lifetime of such data need to be managed effectively without affecting the performance
- Scopes in rails application
 - Request
 - Session
 - Application
 - Flash



Scopes - Request

- The data that comes from the client
 - params
- The data that is passed from controller to view directly (Model)
 - When a controller returns data (model)
- once the view is sent to the browser /client we cannot access the data
 - They live for only one request
- This data can be added as part of redirect
 - browser will send them again as part of second request
 - redirect of index action



- object and values put in session scope will live as long as the client (browser) is in conversation with the controllers in the application
- They work across controllers and multiple requests
- Session must be invalidated using `session.invalidate()` method call to destroy these objects
- Putting a user into session (normally done at login)

```
session.user = userInstance
```

Or

```
session['user'] = userInstance
```

- Other candidates are shopping carts, search results for multiple filters



Scope - Application

- Common to all clients of the application
- Lives as long as the application runs
- a.k.a servletContext
- You can put anything here
- cache some fixed data from the database
 - List of states
 - List of genders
 - Currency conversion rates

```
servletContext['USDtoINR'] = 55.34  
servletContext.USDtoINR = 55.34
```



Scope - Flash

- We have seen this already
- Specific to Grails
- You can put anything in the flash scope
- Lives for 2 requests (used in redirect)



Message function

- Avoids hard coding text strings in the application
- can be used for internationalization - i18n
- Flavors - static text, dynamically replaceable text
- Look at file message.properties in i18n folder
- Static text
 - A text with no place holders
`default.home.label=Home`
- Replaceable
 - Text with placeholders
`default.created.message={0} {1} created`
 - Placeholders {0} & {1} are replaced with message() function



Message function

- Simple

```
message(code: 'default.home.label')
```

- With default - if circle.label not found in message.properties

```
message(code: 'circle.label', default: 'Circle')
```

- With args - Replaceable

```
message(code: 'default.created.message',  
        args: ['Circle', circleInstance.id])
```

- There is a tag <g:message> with equivalent functionality in GSP

- In fact all tags of GSP can be used in controllers as method calls



- A method to create a JSON out of error messages
- private method in CircleController

```
private def error2json(circleInstance) {  
    def msgs = [:]  
  
    eachError(bean : circleInstance) {  
        msgs << ["${it.getField()}" : message(error : it)]  
    }  
    return msgs  
}
```



- Put this in save action

```
def save() {  
    def circleInstance = new Circle()  
    circleInstance.properties = request.JSON  
    def responseBody = [:]  
    if(circleInstance.save(flush:true)) {  
        responseBody.id = circleInstance.id  
        responseBody.message = 'Circle created successfully'  
        response.status = 200  
    }  
    else {    //some validation error  
        responseBody = error2json(circleInstance)  
        response.status = 400  
    }  
    render responseBody as JSON  
}
```



- Circle has a Owner property of type User
- When a Circle is converted to JSON we get only the user_id
- If we want to get the owner name also in the Circle then we need to Register a Marshaller
- We can do it in BootStrap.groovy
- In fact we can have multiple BootStrap files (any groovy file with a name ending with BootStrap)
- Let us create a new BootStrap for JSON marshalling



Customise JSON Generation

```
class JsonMarshallerBootStrap {  
    grails.converters.JSON.registerObjectMarshaller(Circle) {  
        Circle circle ->  
        def output = [:]  
        output.id = circle.id  
        output.version = circle.version  
        output.name = circle.name  
        output.description = circle.description  
        output.owner = [id:circle.owner.id,  
                        name:circle.owner.username]  
        return output  
    }  
}
```



Grails Application Development - Controllers

Services

Services - Why do we need?

- Service Layer - Part of JEE application Stack
- Abstracted business Layer with centralized business logic
- logic involves two or more domain classes - that wont fit in a controller
- Help avoid controller bloating with logic/code
 - Lean controllers - easy to maintain
- An application level API
- Can be injected where ever we want



What is a service?

- So far we have been creating circles with an owner
- Cant we make owner automatically become a member of the circle that he/she created?
- Consider putting this code in the “save” action of the controller
- You need to first save the circle and then add the membership - 2 pieces of logic & out of place for create circle action



Creating a service

- New -> Service menu or
- `grails create-service` command with a name (with package)
- `grails create-service com.ardhika.learn.Circle`
- name is given without Service suffix (even while using the menu of GGTS)
- A class by name `CircleService.groovy` will be created with a default method

```
package com.ardhika.learn

class CircleService {

    def serviceMethod() {

    }

}
```



Creating a Service

- Rename the method and write this code

```
boolean createCircle(Circle circle) {  
    if(!circle.save()) return false  
    if(!Membership.subscribe(circle.owner, circle))  
        return false  
    return true  
}
```

- Here owner of the circle is added as a member using the subscribe method of the Membership domain class
- You can also use the circle.addToMembers method



Consuming a Service

- Done with Spring Dependency Injection
- Declare a variable with name circleService
- Note the came casing
- variable name is the same as Service class name
- Top of the CircleController class add this
`CircleService circleService`
- Spring will infer this and inject an object of type CircleService
- No need to create the object with `new CircleService()`
- Time to rewrite the create action of CircleController



- New code for create action

```
def save() {  
    def circleInstance = new Circle(params)  
    //if (!circleInstance.save(flush: true)) {  
    if(!circleService.createCircle(circleInstance)) {  
        render(view: "create",  
                model: [circleInstance: circleInstance])  
        return  
    }  
  
    flash.message = message(code:  
                            'default.created.message',  
                            args: [message(code: 'circle.label',  
                                            default: 'Circle'), circleInstance.id])  
    redirect(action: "show", id: circleInstance.id)  
}
```



Transaction

- By default rails service methods are transactional
- if you want to switch it off use

```
class CircleService {  
  static transactional = false  
  
  //service methods  
}
```
- Transactions work only with dependency injection
- If an exception is thrown in the service method due to validation or any other error transaction will be rolled back



Grails Application Development - Controllers

Security

Securing Controllers

- Check the myCircles() action of the CircleController
- It just accesses the user object stored in session
- What will happen if you invoke circles/myCircles without logging in first?
- Present code will throw exceptions & fail

```
def myCircles() {  
    def userId=session.loggedInUser.id  
    def user=User.get(userId)  
  
    render user.circles as JSON  
}
```



- If the user has not logged in we should put the user to login

```
def myCircles(){
  def loggedInUser = session.loggedInUser
  if(!loggedInUser) {
    //send error not logged in status 403
  }
  else {
    def userId=loggedInUser.id
    def user=User.get(userId)
    render user.circles as JSON
  }
}
```

- Doing the same for all actions in all controllers? **Tedious!**



Securing Controllers - Using interceptors

- Instead securing actions - Secure Controllers
- Intercept every request that gets into the actions
- Designate a function to execute while intercepting
- You can intercept before or after a request

```
//set before interceptor to execute checkAuth function
def beforeInterceptor = [action: this.&checkAuth]
//Define checkAuth function
def checkAuth() {
    if(!session.loggedInUser) {
        //render response with status 403
        return false
    }
    return true
}
```

- Put this in CircleController



Securing Controllers - Using interceptors

- We need to do this in every controller
- If we do this in the user controller this will intercept requests for login, authenticate, create(register) and save(register)
- Obviously you don't have to login to register yourself
- Fortunately you can have an exclusion list
- In UserController you can write

```
def beforeInterceptor = [action:this.&checkAuth,
                        except:['login', 'logout', 'create', 'save']]
```
- But there is a method better than this!
- Define filters at the app level



Securing at app level - Filters

- Create a new -> Filter with name LearnSecurity or (grails command create-filters)
- This creates LearnSecurityFilters class **conf** folder

```
class LearnSecurityFilters {  
    def filters = {  
        all(controller:'*', action:'*') {  
            before = {  
            }  
            after = { Map model ->  
            }  
            afterView = { Exception e ->  
            }  
        }  
    }  
}
```



- before
 - code executes before action
- after
 - code executes after action but before view rendering
 - Do something to the **model** if needed
- afterView
 - code executes after rendering the view
 - Can handle **exceptions** if there are any



Implementing a Filter

- There is a default filter for all controllers & actions
- Let us add code in the **before** block
- But exclude check for User controllers actions for login and register

```
before = {  
  if(!session.loggedInUser &&  
    !((controllerName == 'user') &&  
      (actionName == 'login' || actionName == 'authenticate' ||  
        actionName == 'create' || actionName == 'save')) {  
    redirect(controller:"user", action:"login")  
    return false  
  }  
}
```



Implementing a Filter

- Filters syntax helps you to make it neat without those clumsy conditions
- Refer Grails Reference for Filters
- All controllers and actions except the following

```
allExceptLoginAndRegister(controller: 'user',  
                           action: ' (login|authenticate|create|save) ',  
                           invert: true) {  
  before = {  
    if(!session.loggedInUser) {  
      //render response with status 403  
      return false  
    }  
  }  
}
```



- Codec as used to transform a string and doing the reverse
 - encodeAsHTML() & decodeHTML()
- There are variety of in built codecs in Grails
- Play a major part in security
- can build a custom codec for password hashing/encryption

- Let us try a technique - reverse and store password
- Codec is a groovy class with name ending in Codec stored in the utils folder



- ReversedPasswordCodec.groovy in grails-app/utils

```
class ReversedPasswordCodec {  
    static encode = { str->  
        return str.reverse()  
    }  
  
    static decode = { str->  
        return str.reverse()  
    }  
}
```

- Now while saving the user encode the password

```
userInstance.password.encodeAsReversedPassword()
```

- Can be done in domain class [beforeInsert](#) also
- Similarly encode the password in the authenticate action



Security Plugins

- Can be installed using `install-plugin` grails command
- Plugins are like mini projects with a set of components that we can reuse
- Spring Security Plugin
 - Comes with User, Role userRole domain classes
 - Login and logout actions and views
 - Tag library for checking the roles (securing portions of views)



Thank You!



Bala Sundarasamy
bala@ardhika.com

www.ardhika.com