



Groovy & Grails: Desarrollo rápido de aplicaciones

Sesión 10: Dominios y servicios (II)



Dominios y servicios (II)

- Interactuar con la base de datos
- Servicios



Interactuar con la base de datos

- Consultas dinámicas de GORM
- Consultas HQL de Hibernate
- Consultas Criteria de Hibernate



Interactuar con la base de datos

- Los métodos habituales presentes en las clases de dominio son *get()*, *delete()* y *save()*
- Estos métodos no hemos tenido que crearlos nosotros mismos
- Grails se encarga de interceptar las llamadas a estos métodos y responder por ellos



Interactuar con la base de datos

```
void testCRUDOperation() {
    //Creamos el nuevo usuario
    def usuarioTemp = new Usuario(login:'otrousuuario',
    password: 'otropasswd', nombre: 'Otro', apellidos: 'Usuario',
    tipo: 'profesor')
    //Tratamos de persistirlo en la base de datos
    usuarioTemp.save()
    //Comprobamos que el número total de usuarios coincide con
    //lo que esperamos. Ten en cuenta que el método setUp() ha
    //añadido otro usuario
    assert Usuario.count() == 7
    //Obtenemos el identificador del usuario recién creado
    def usuarioId = usuarioTemp.id
    ....
}
```



Interactuar con la base de datos

```
void testCRUDOperation() {  
    ....  
    //Tratamos de actualizar los datos del nuevo usuario  
    usuarioTemp.password = 'nuevopasswd'  
    //Comprobamos que se ha actualizado el password  
    usuarioTemp = Usuario.get(usuarioId)  
    assert "nuevopasswd" == usuarioTemp.password  
    assert "Otro" == usuarioTemp.nombre  
    //Ahora eliminamos el usuario  
    usuarioTemp.delete()  
    //Y comprobamos que se elimina correctamente  
    assert null == Usuario.get(usuarioId)  
    assert Usuario.count() == 6  
}
```



Interactuar con la base de datos

```
grails test-app Usuario
```

- El nuevo test se pasa correctamente
- El método *get()* sólo permite seleccionar un registro de la base de datos
- Necesitamos más formas de seleccionar registros



Consultas dinámicas de GORM

- **Contadores**

- Para conocer el número de registros de la clase de dominio *Usuario* podemos utilizar el método *count()*

```
Usuario.count()
```

- Si necesitamos saber el número de *administradores* del sistema

```
Usuario.countByTipo("administrador")
```




Consultas dinámicas de GORM

- **Contadores**

- Permite anidar criterios de búsqueda

```
Usuario.countByTipoOrTipo("administrador", "bibliotecario")
```

- Podemos emplear tanto criterios *AND* como *OR*

```
Usuario.countByTipoAndNombre("administrador", "Francisco José")
```



Consultas dinámicas de GORM

- **Consultas que devuelven un solo registro**
 - Podemos utilizar la función de GORM *findBy()*

```
Usuario.findByTipo("administrador")
```

- Podemos emplear tanto criterios *AND* como *OR*

```
Usuario.findByTipoAndNombre("administrador","Francisco José")
```



Consultas dinámicas de GORM

- **Consultas que devuelven un solo registro**
 - Si todos los criterios de nuestra consulta deben cumplirse, podemos utilizar la función *findWhere()*
 - Se le pasa como parámetro un *mapa* con las propiedades y los valores que deben cumplir los registros devueltos

```
Usuario.findWhere(["tipo":"administrador", "nombre":"Francisco José"])
```



Consultas dinámicas de GORM

- **Consultas que devuelven un solo registro**
 - El método *findWhere()* es especialmente útil en aquellos casos en los que nos sea devuelto un mapa de propiedades de otro procedimiento
 - Los métodos *findBy()* y *findWhere()* sólo devuelven un registro de la clase de dominio correspondiente



Consultas dinámicas de GORM

- **Consultas que devuelven varios registros**
 - El método *findAllBy()* es similar al método *findBy()* con la salvedad que ahora se devuelven todos los registros que cumplan las condiciones impuestas

```
Usuario.findAllByTipo("socio")
```

- Podemos emplear tanto criterios *AND* como *OR*

```
Usuario.findAllByTipoOrTipo("socio","profesor")
```



Consultas dinámicas de GORM

- **Consultas que devuelven varios registros**
 - El método *findAllWhere()* es similar al método *findWhere()* con la salvedad que ahora se devuelven todos los registros que cumplan las condiciones impuestas

```
Usuario.findAllWhere(["tipo":"profesor", "apellidos":"Pica Piedra"])
```



Consultas dinámicas de GORM

- **Consultas que devuelven varios registros**
 - El método *getAll()* es similar al método *get()*
 - El método *getAll()* recibe como parámetro una lista de identificadores

```
Usuario.getAll([1,3,5])
```



Consultas dinámicas de GORM

- **Consultas que devuelven varios registros**
 - El método *list()* devuelve todas las instancias de una determinada clase de dominio

```
Usuario.list()
```




Consultas dinámicas de GORM

- **Consultas que devuelven varios registros**
 - El método *list()* puede recibir una serie de parámetros para filtrar los registros devueltos
 - *max*, permite indicar un número máximo de registros devueltos
 - *offset*, los registros devueltos deben empezar desde una posición determinada. En combinación con el parámetro *max* permite realizar paginación sobre los resultados
 - *sort*, indica una propiedad de la clase de dominio por la que ordenar los registros devueltos
 - *order*, ascendente o descendente



Consultas dinámicas de GORM

- **Otros criterios de búsqueda**
 - Hasta ahora las propiedades debían coincidir completamente con el valor pasado por parámetro
 - Podemos utilizar caracteres comodines con la función *findAllBy__Like()*

```
Usuario.findAllByNombreLike("%fran%")
```



Consultas dinámicas de GORM

- **Otros criterios de búsqueda**
 - También podemos encadenar varios criterios

```
Usuario.findAllByNombreLikeAndApellidosLike(“%fran%”, “%arc%”)
```



Consultas dinámicas de GORM

- Otros criterios de búsqueda

Método	Descripción
<i>InList</i>	Devuelve aquellos registros en los que el valor de la propiedad dada coincida con cualquiera de los elementos de la lista pasada por parámetro
<i>LessThan</i>	Devuelve aquellos registros en los que el valor de la propiedad dada sea menor que el valor pasado por parámetro
<i>LessThanEquals</i>	Devuelve aquellos registros en los que el valor de la propiedad dada sea menor o igual que el valor pasado por parámetro
<i>GreaterThan</i>	Devuelve aquellos registros en los que el valor de la propiedad dada sea mayor que el valor pasado por parámetro
<i>GreaterThanEquals</i>	Devuelve aquellos registros en los que el valor de la propiedad dada sea mayor o igual que el valor pasado por parámetro



Consultas dinámicas de GORM

- Otros criterios de búsqueda

Método	Descripción
<i>GreaterThanEquals</i>	Devuelve aquellos registros en los que el valor de la propiedad dada sea mayor o igual que el valor pasado por parámetro
<i>Like</i>	Es equivalente a una expresión <i>LIKE</i> en una sentencia SQL
<i>ILike</i>	Similar al método <i>Like</i> salvo que en esta ocasión <i>case insensitive</i>
<i>NotEqual</i>	Devuelve aquellos registros en los que el valor de la propiedad dada no sea igual al valor pasado por parámetro



Consultas dinámicas de GORM

- Otros criterios de búsqueda

Método	Descripción
<i>Between</i>	Devuelve aquellos registros en los que el valor de la propiedad dada se encuentre entre los dos valores pasados por parámetro. Necesita de dos parámetros
<i>IsNull</i>	Devuelve aquellos registros en los que el valor de la propiedad dada no sea nula. No se necesita ningún argumento.
<i>IsNull</i>	Devuelve aquellos registros en los que el valor de la propiedad dada sea nula



Consultas dinámicas de GORM

- **Resumen de métodos dinámicos de GORM**

Método	Descripción
<i>count()</i>	Devuelve el número total de registros de una determinada clase de dominio
<i>countBy()</i>	Devuelve el número total de registros de una determinada clase de dominio que cumplan una serie de requisitos encadenados tras el nombre del método
<i>findBy()</i>	Devuelve el primer registro encontrado de una determinada clase de dominio que cumpla una serie de requisitos encadenados tras el nombre del método
<i>findWhere()</i>	Devuelve el primer registro encontrado de una determinada clase de dominio que cumpla una serie de requisitos pasados por parámetro en forma de mapa



Consultas dinámicas de GORM

- **Resumen de métodos dinámicos de GORM**

Método	Descripción
<i>findAllBy()</i>	Devuelve todos los registros encontrados de una determinada clase de dominio que cumplan una serie de requisitos encadenados tras el nombre del método
<i>findAllWhere()</i>	Devuelve todos los registros encontrados de una determinada clase de dominio que cumplan una serie de requisitos pasados por parámetro en forma de mapa
<i>getAll()</i>	Devuelve todos los registros de una determinada clase de dominio cuyo identificador coincida con cualquier de los pasados parámetro en forma de lista



Consultas dinámicas de GORM

- **Resumen de métodos dinámicos de GORM**

Método	Descripción
<i>list()</i>	Devuelve todos los registros de una determinada clase de dominio. Acepta los parámetros <i>max</i> , <i>offset</i> , <i>sort</i> y <i>order</i>
<i>listOrderBy()</i>	Devuelve todos los registros de una determinada clase de dominio ordenador por un criterio. Acepta los parámetros <i>max</i> , <i>offset</i> y <i>order</i>



Consultas HQL de Hibernate

- HQL es el lenguaje propio de Hibernate utilizado para realizar consultas
- Tenemos los métodos ya conocidos *find()* y *findAll()*, más uno nuevo llamado *executeQuery()*



Consultas HQL de Hibernate

- El método *find()* devuelve el primer registro de la consulta HQL pasada por parámetro

```
def sentenciaHQL1 = Usuario.find("From Usuario as u")  
assert sentenciaHQL1.tipo == "administrador"
```



Consultas HQL de Hibernate

- El método *findAll()* devuelve todos los registros de la consulta HQL pasada por parámetro

```
def hqlsentence2 = Usuario.findAll("from Usuario as u where u.tipo='socio'")
assert hqlsentence2.size() == 2
```

```
def hqlsentence3 = Usuario.findAll("from Usuario as u where u.tipo=?", ["socio"])
assert hqlsentence3.size() == 2
```

```
def hqlsentence4 = Usuario.findAll("from Usuario as u where u.tipo=:tipo",
[tip:"socio"])
assert hqlsentence4.size() == 2
```



Consultas HQL de Hibernate

- HQL permite utilizar los parámetros *max*, *offset*, *sort* y *order* para afinar aún más la búsqueda
- En los métodos *find()* y *findAll()* obtenemos todas las columnas de la tabla en cuestión
- Con el método *executeQuery()* podemos especificar que columnas deseamos

```
Usuario.executeQuery("select nombre, apellidos from Usuario u where  
u.tipo='administrador'")
```



Consultas Criteria de Hibernate

- Criteria es un API de Hibernate diseñado específicamente para la realización de consultas complejas
- En Grails criteria está basado en un *Builder de Groovy*
- El siguiente código obtiene un listado de todas los *préstamos* de los últimos 10 días



Consultas Criteria de Hibernate

```
void testCriteria() {  
    def c = Operacion.createCriteria()  
    def resultado = c{  
        between("fechaInicio",new Date()-10,new Date())  
        eq("tipo","prestamo")  
        maxResults(15)  
        order("fechaInicio","desc")  
    }  
    assert resultado.size()==1  
}
```



Consultas Criterias de Hibernate

- Criterias dispone de una serie de *criterios* para ser utilizados en este tipo de consultas

Criterio	Ejemplo
<code>between()</code>	<code>between("fechaInicio", new Date()-10, new Date())</code>
<code>eq()</code>	<code>eq("tipo", "prestamo")</code>
<code>eqProperty()</code>	<code>eqProperty("fechaInicio", "fechaFin")</code>
<code>gt()</code>	<code>gt("fechaInicio", new Date()-5)</code>
<code>gtProperty()</code>	<code>gtProperty("fechaInicio", "fechaFin")</code>
<code>ge()</code>	<code>ge("fechaInicio", new Date()-5)</code>
<code>geProperty()</code>	<code>geProperty("fechaInicio", "fechaFin")</code>



Consultas Criterias de Hibernate

Criterio	Ejemplo
<i>idEq()</i>	<code>idEq(1)</code>
<i>ilike()</i>	<code>ilike("nombre", "fran%")</code>
<i>in()</i>	<code>'in'("edad",[18..65])</code>
<i>isEmpty()</i>	<code>isEmpty("descripcion")</code>
<i>isNotEmpty()</i>	<code>isNotEmpty("descripcion")</code>
<i>isNull()</i>	<code>isNull("descripcion")</code>
<i>isNotNull()</i>	<code>isNotNull("descripcion")</code>
<i>lt()</i>	<code>lt("fechaInicio",new Date()-5)</code>
<i>ltProperty()</i>	<code>ltProperty("fechaInicio","fechaFin")</code>



Consultas Criterias de Hibernate

Criterio	Ejemplo
<i>le()</i>	<code>le("fechaInicio",new Date()-5)</code>
<i>leProperty()</i>	<code>leProperty("fechaInicio","fechaFin")</code>
<i>like()</i>	<code>like("nombre","Fran%")</code>
<i>ne()</i>	<code>ne("tipo","prestamo")</code>
<i>neProperty()</i>	<code>neProperty("fechaInicio","fechaFin")</code>
<i>order()</i>	<code>order("nombre","asc")</code>
<i>rlike()</i>	<code>rlike("nombre",/Fran.+/)</code>
<i>sizeEq()</i>	<code>sizeEq("nombre",10)</code>
<i>sizeGt()</i>	<code>sizeGt("nombre",10)</code>



Consultas Criterias de Hibernate

Criterio	Ejemplo
<i>sizeGe()</i>	<code>sizeGe("nombre",10)</code>
<i>sizeLt()</i>	<code>sizeLt("nombre",10)</code>
<i>sizeLe()</i>	<code>sizeLe("nombre",10)</code>
<i>sizeNe()</i>	<code>sizeNe("nombre",10)</code>



Consultas Criterias de Hibernate

- Los criterios de búsqueda se pueden agrupar en bloques lógicos *AND*, *OR* y *NOT*

```
and {
    between("fechaInicio", new Date()-10, new Date())
    eq("tipo", "prestamo")
}
or {
    between("fechaInicio", new Date()-10, new Date())
    eq("tipo", "prestamo")
}
not {
    between("fechaInicio", new Date()-10, new Date())
    eq("tipo", "prestamo")
}
```



Consultas Criterias de Hibernate

- Obtener información en función de clases asociadas es muy sencillo con *criteria*

```
def c2 = Operacion.createCriteria()
def resultado2 = c2 {
    usuario {
        eq("tipo","socio")
    }
}
```

- Simplemente agrupamos dentro del bloque *usuario* con las condiciones que se deben cumplir



Consultas Criterias de Hibernate

- Criterias también tiene la posibilidad de utilizar operaciones agrupadas del estilo *distinct*, *min* o *max* en lo que se conoce como *proyecciones*
- Para saber cuantos usuarios distintos tienen operaciones asignadas

```
def c3 = Operacion.createCriteria()
def numeroUsuarios = c3.get {
    projections {
        countDistinct('usuario')
    }
}
```



Consultas Criterias de Hibernate

- Métodos para agrupar resultados

Nombre	Ejemplo
<i>property()</i>	<code>property("fechaInicio")</code>
<i>distinct()</i>	<code>distinct("fechaInicio", "fechaFin")</code>
<i>avg()</i>	<code>avg("edad")</code>
<i>count()</i>	<code>count("nombre")</code>
<i>countDistinct()</i>	<code>countDistinct("tipo")</code>
<i>groupByProperty()</i>	<code>groupByProperty("usuario")</code>
<i>max()</i>	<code>max("edad")</code>
<i>min()</i>	<code>min("edad")</code>
<i>sum()</i>	<code>sum("ingresos")</code>
<i>rowCount()</i>	<code>rowCount()</code>



Servicios

- En el patrón *Modelo Vista Controlador*, los servicios deben ocuparse de la lógica de negocio
- Un error muy común es acumular demasiado código en los controladores
- Si implementáramos servicios, nuestra aplicación sería más fácil de mantener



Servicios

- Los servicios en Grails son clases cuyo nombre termina con la palabra *Service* y se ubican en el directorio *grails-app/services*
- Vamos a crear un servicio para dar de alta nuevos usuarios
- Creamos un nuevo servicio

```
grails create-service usuario
```



Servicios

- Se crea un servicio

```
class UsuarioService {  
    boolean transactional = true  
    def serviceMethod() {  
    }  
}
```

- Y un test unitario

```
import grails.test.*  
class UsuarioServiceTests extends GrailsUnitTestCase {  
    ....  
}
```



Servicios

- Si un servicio es *transaccional*, cuando se produce cualquier error o excepción, las operaciones realizadas serán desechadas
- Vamos a crear un método en el servicio para dar de alta nuevos usuarios
- El método recibirá un mapa con todos los datos del usuario



Servicios

```
Usuario altaUsuario(params) {  
    def u = new Usuario(params)  
  
    //Comprobamos los datos introducidos con las restricciones  
    //de la clase de dominio Usuario  
    if (u.validate()){  
        //Almacenamos en la base de datos  
        u.save()  
    }  
    return u  
}
```



Servicios

- Creamos una nueva instancia de la clase Usuario con los parámetros recibidos
- Se validan las restricciones con el método *validate()*
- Si no se produce ningún error, se almacena el usuario en la base de datos



Servicios

- Si se produce algún error, la propiedad *errors* del objeto *u* contendrá los errores producidos
- Se devuelve al controlador los datos del usuario para que el controlador decida que hacer con ellos
- Debemos modificar el controlador de la clase Usuario para que utilice el servicio recién creado



Servicios

```
def usuarioService
....
def save = {
    def u = usuarioService.altaUsuario(params)
    if(!u.hasErrors()) {
        flash.message = "Usuario ${u.nombre} ${u.apellidos} created"
        redirect(action:show,id:u.id)
    }
    else {
        render(view:'create',model:[usuarioInstance:u])
    }
}
```



Servicios

- Por defecto, los servicios son de tipo *singleton*, es decir, sólo existe una instancia de la clase que se inyecta a todos los artefactos
- Esto supone que no podemos guardar información *privada* puesto que todos los controladores verían esa información



Servicios

- Podemos solucionar este problema definiendo una variable *scope* con cualquiera de estos valores

Valor	Descripción
<i>prototype</i>	Se crea una nueva instancia por cada servicio inyectado
<i>request</i>	Se crea una nueva instancia por cada solicitud HTTP
<i>flash</i>	La instancia es accesible por la solicitud actual y la siguiente
<i>flow</i>	Se crea una instancia nueva en cada flujo
<i>conversation</i>	La instancia será visible para el flujo actual y sus subflujos
<i>session</i>	Se crea una instancia nueva del servicio para cada sesión
<i>singleton</i>	Sólo existe una instancia del servicio



Servicios

- Si queremos modificar el ámbito del servicio

```
class UsuarioService{  
    static scope = 'session'  
    .....  
}
```



Servicio de mensajería

- Se refiere al envío de correos electrónicos
- Se utilizará este servicio para enviar notificaciones a los usuarios
- Debemos instalar el plugin *Mail* para facilitar la labor

```
grails install-plugin mail
```



Servicio de mensajería

- Debemos añadir en el archivo *grails-app/conf/Config.groovy* una variable con el servidor SMTP que vamos a utilizar en el envío

```
grails.mail.host = "mail.ua.es"
```

- Creamos el nuevo servicio que llamaremos *Notificador*

```
grails create-service Notificador
```



Servicio de mensajería

- El único método de este servicio será el encargado de enviar los emails

```
class NotificadorService {
    boolean transactional = false
    def mailService
    def mandarMails(email) {
        mailService.sendMail {
            to email
            from "fgarcia@ua.es"
            subject "Estoy probando mi servicio de envío de emails"
            body "Disculpa las molestias"
        }
    }
}
```



Servicio de mensajería

- Para probar el servicio, creamos un nuevo método en el controlador de la clase Usuario para enviar un mail de prueba

```
def enviarEmails = {  
    notificadorService.mandarMails("fgarcia@ua.es")  
    render "El email ha sido enviado correctamente"  
}
```

- <http://localhost:8080/biblioteca/usuario/enviarEmails>