Dominios y servicios (II)

Índice

En la primera sesión sobre Grails, ya vimos como crear una completa aplicación web capaz incluso de persistir los datos en una base de datos, sin embargo, toda la información sobre como Grails se encarga de esta parte quedó muy en el aire, así que en esta sesión vamos a ver como Grails es capaz de trabajar con las bases de datos. Se asumirán algunos conocimientos mínimos de lenguaje SQL, aunque también es posible trabajar con Grails sin tener apenas idea de SQL.

Para empezar, veremos como Grails dispone de varias formas de acceder a una base de datos y realizar consultas sobre ellas. En primer lugar, repasaremos como realizar las simples operaciones CRUD (`create, read, update y delete()`). Posteriormente, veremos como GORM nos muestra la potencia de Groovy a la hora de crear DSL's para realizar consultas a la base de datos a partir de los nombres de los métodos utilizados, tal y como vimos en la sesión anterior con el método `findAllByTipoAndUsuario()`.

Por último, comprobaremos como en ocasiones la única forma de realizar las consultas a la base de datos es utilizando el lenguaje propio de Hibernate (*HQL - Hibernate Query Language*).

1. Interactuar con la base de datos

Cuando interactuamos con una base de datos, lo primero que debemos conocer es como realizar las operaciones básicas en cualquier aplicación (CRUD). Si echamos un vistazo al código del controlador de la clase de dominio *Usuario*, detectaremos la presencia de los métodos `get()`, `delete()` y `save()`. También podemos comprobar como no existe ninguno de estos tres métodos implementados en la clase de dominio *Usuario*. Entonces, ¿cómo es capaz Grails de persistir la información en la base de datos si estos tres métodos no existen?

La respuesta es sencilla. Grails se encarga de interceptar las llamadas a estos métodos y responder por ellos. Veamos un ejemplo de como funcionan estos tres métodos implementando un nuevo test de integración para la clase de dominio *Usuario*

```
void testCRUDOperation() {
    //Creamos el nuevo usuario
    def usuarioTemp = new Usuario(login:'otrousuario',
    password: 'otropasswd',
    nombre: 'Otro',
    apellidos: 'Usuario',
    tipo: 'profesor',
    email: 'miemail@ua.es')

    //Tratamos de persistirlo en la base de datos
    usuarioTemp.save()

    //Comprobamos que el numéro 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 recien creado
    def usuarioId = usuarioTemp.id
```

```
    //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
}
```

Si ejecutamos ahora `grails test-app biblioteca.Usuario`, comprobaremos que el test funciona correctamente, con lo que los métodos `get()`, `save()` y `delete()` funcionan tal y como esperábamos.

Ahora bien, con el método `get()`, sólo vamos a poder seleccionar un registro de la base de datos que concuerde con el identificador de la clase de dominio en cuestión. Por supuesto, necesitaremos más formas de obtener estos registros a partir de otras propiedades.

Las diversas formas de realizar consultas a la base de datos son:

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

1.1. Consultas dinámicas de GORM

Hasta el momento, para crear las operaciones básicas (CRUD) de cualquier aplicación, no hemos necesitado más que recuperar los datos de una determinada instancia de una clase de dominio a partir de su identificador (clave primaria). Pero, creer que esto sería suficiente para desarrollar una completa aplicación sería de ilusos, así que vamos a profundizar en otro tipo de consultas a la base de datos.

Contadores

Si deseamos conocer el número total de registros de una clase de dominio, podemos utilizar el método `count()`, tal y como ya hemos visto en el test de ejemplo anterior `assert Usuario.count()==7`. Pero, ¿qué pasa si quisiéramos saber cuantos usuarios de tipo *administrador* tenemos en nuestro sistema?

GORM permite afinar el método `count()` para devolver sólo el número total de registros que cumplen una determinada condición. Por ejemplo, para conocer cuantos administradores hay en el sistema podemos utilizar el método `Usuario.countByTipo("administrador")`.

Pero la potencia de GORM como lenguaje DSL para realizar consultas a la base de datos

no se queda ahí, sino que va un paso más allá y permite anidar criterios de búsqueda de tipo `AND` y `OR`. Si por ejemplo, necesitáramos conocer cuantos usuarios de tipo *administrador* y *bibliotecario* hay en nuestra aplicación, podríamos escribir `Usuario.countByTipoOrTipo("administrador","bibliotecario")`.

No hace falta comentar, que al igual que el segundo criterio de búsqueda en el ejemplo anterior ha sido el *tipo*, podríamos haber utilizado cualquier otro de entre las propiedades de la clase de dominio *Usuario* y que, de igual forma, también podríamos haber empleado una búsqueda de tipo `AND` para buscar el número de registros que cumplan ambas condiciones.

Consultas que devuelven un solo registro

En cualquier aplicación, es necesario mostrar la información de una determinada instancia de una clase de dominio que cumpla una serie una serie de condiciones. Con GORM esto es posible de dos formas que vamos a ver a continuación. La primera de ellas es mediante el método `findBy()`. Este método, al igual que hacíamos con el método `count()` permite añadir una serie de propiedades de la clase de dominio encadenadas por las palabras reservadas `AND` y `OR` para devolver un conjunto de registros.

Siguiendo con el ejemplo anterior, si escribimos `Usuario.findByTipo("administrador")`, la aplicación nos devolvería el primer registro que cumpla que la propiedad *tipo* sea *administrador*. ¿Y si quisiéramos mostrar el administrador cuyo nombre sea *Francisco José*? Pues siguiendo el mismo procedimiento que con el método `count()`, podríamos tener `Usuario.findByTipoAndNombre("administrador","Francisco José")`.

Este método es muy útil, sobre todo si se desea mezclar criterios de búsqueda `AND` y `OR`, pero si necesitamos que en nuestra consulta se cumplan todos y cada uno de los criterios, podemos emplear el método `findWhere()`, al cual se le pasa como parámetro un mapa con las propiedades de la clase de dominio sobre las que deseamos realizar las consultas y su valor. Para el ejemplo anterior podríamos tener `Usuario.findWhere(["tipo":"administrador", "nombre":"Francisco José"])`.

El método `findWhere()` puede ser especialmente útil en aquellos casos en los que nos sea devuelto un mapa de propiedades de una clase de dominio de cualquier llamada a otro procedimiento. De esta forma, simplemente tendríamos que pasar el valor devuelto en esa llamada como parámetro al método `findWhere()`.

Recuerda que los métodos `findBy()` y `findWhere()` sólo devuelven un registro de la clase de dominio correspondiente.

Consultas que devuelven varios registros

Con los métodos anteriores `findBy()` y `findWhere()` simplemente éramos capaces de obtener un registro de la clase de dominio implicada, lo cual no suele lo más común en las aplicaciones, donde normalmente se necesita obtener una serie de registros para

posteriormente mostrárselos al usuario. A continuación, vamos a ver varias formas para la obtención de este conjunto de registros.

El método `findAllBy()` es similar al método `findBy()` con la única diferencia de que en este caso se devuelven todos los registros que cumplan las condiciones impuestas en el método. Si necesitáramos mostrar un listado con aquellos usuarios de tipo *socio* de nuestra aplicación podríamos escribir `Usuario.findAllByTipo("socio")`.

De igual forma que con el método `findBy()`, vamos a poder encadenar diferentes criterios para filtrar esa búsqueda de registros. Por ejemplo, si en un listado debemos mostrar aquellos usuarios de tipo *socio* y *profesor* de nuestro sistema, deberíamos escribir `Usuario.findAllByTipoOrTipo("socio","profesor")`.

Análogamente al método `findWhere()`, también disponemos del método `findAllWhere()`, que nos devolverá todos los registros que cumplan las restricciones pasadas por parámetro en forma de mapa. Si quisiéramos obtener todos los profesores de nuestro sistema que tengan como apellidos *Pica Piedra* podríamos tener `Usuario.findAllWhere(["tipo":"profesor", "apellidos":"Pica Piedra"])`. Recuerda que, tanto en el método `findWhere()` como en `findAllWhere()` todas las condiciones pasadas como parámetro deben cumplirse, con lo que `Usuario.findAllByTipoAndApellidos("profesor","Pica Piedra")` sería idéntico al método `Usuario.findAllWhere(["tipo":"profesor","apellidos":"Pica Piedra"])`.

Sigamos conociendo nuevos métodos para obtener varios registros. El siguiente método que vamos a ver podríamos decir que sería el análogo al método `get()` que ya vimos anteriormente. El método `getAll()` acepta como parámetro una lista de valores para devolver aquellas instancias que tengan como identificador cualquiera de estos valores. Por ejemplo, si quisiéramos obtener aquellos *usuarios* con identificador 1, 3 y 5 podríamos escribir `Usuario.getAll([1,3,5])`.

El siguiente método que vamos a ver, nos permite obtener todas las instancias de una determinada clase de dominio. El método `list()` es posiblemente el método más básico de todos los que podemos encontrar en GORM. Para obtener todos los usuarios de nuestro sistema podríamos tener `Usuario.list()`.

El método `list()` puede recibir una serie de parámetros para filtrar los registros devueltos y la forma en como se devuelven. Estos parámetros son:

- `max`, que permite indicar un número máximo de registros devueltos
- `offset`, que indicará que los registros devueltos empiecen desde una determinada posición. En combinación con el parámetro `max` permitirá realizar paginación sobre los resultados
- `sort`, indica una propiedad de la clase de dominio por la que ordenar los registros devueltos
- `order`, especifica un orden (ascendente o descendente) para los registros devueltos

Podríamos decir que el método `list()` tiene otro método mejorado que nos permite ordenar los registros devueltos por un determinado criterio. El método `listOrderBy()` permite encadenar propiedades de la clase de dominio correspondiente para ordenar los registros devueltos. Si el listado anterior de todos los usuarios del sistema lo necesitáramos ordenado por nombre, podríamos tener `Usuario.listOrderByNombre()`. Este método también acepta los parámetros indicados anteriormente para `list()` con la salvedad del parámetro *sort*, puesto que este valor ya se especifica en el propio nombre del método.

Otros criterios de búsqueda

Hasta el momento, en todos los métodos que hemos visto que permitían buscar registros entre las instancias de una clase de dominio debía coincidir literalmente el valor de la propiedad con el valor pasado por parámetro. Por ejemplo, en la llamada `Usuario.findAllByNombre("Francisco José")`, el sistema busca aquellos usuarios que tengan como nombre *Francisco José*. Pero, ¿cómo podríamos conseguir que el sistema nos devuelva todos los usuarios que tengan en su nombre la cadena *fran*?

Para ello, GORM dispone de otras opciones de búsqueda que permiten afinarla. Por ejemplo, para mostrar todos los usuarios que tengan en su nombre la palabra *cisco*, podríamos tener lo siguiente `Usuario.findAllByNombreLike("%cisco%")`. Pero aún podríamos afinar más esta búsqueda, obteniendo sólo aquellos usuarios que tengan también en su apellidos la cadena *arc* de la siguiente forma `Usuario.findAllByNombreLikeAndApellidosLike("%cisco%", "%arc%")`.

La siguiente tabla muestra todas los posibles métodos de comparación en la búsqueda de registros.

Método	Descripción
InList	Devuelve aquellos registros en los que el valor de la propiedad dada coincida con cualquiera de los elementos de la lista pasada por parámetro
LessThan	Devuelve aquellos registros en los que el valor de la propiedad dada sea menor que el valor pasado por parámetro
LessThanEquals	Devuelve aquellos registros en los que el valor de la propiedad dada sea menor o igual que el valor pasado por parámetro
GreaterThan	Devuelve aquellos registros en los que el valor de la propiedad dada sea mayor que el valor pasado por parámetro
GreaterThanEquals	Devuelve aquellos registros en los que el valor de la propiedad dada sea mayor o igual que el valor pasado por parámetro
Like	Es equivalente a una expresión *LIKE* en una

	sentencia SQL
ILike	Similar al método *Like* salvo que en esta ocasión *case insensitive*
NotEqual	Devuelve aquellos registros en los que el valor de la propiedad dada no sea igual al valor pasado por parámetro
Between	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.
IsNotNull	Devuelve aquellos registros en los que el valor de la propiedad dada no sea nula. No se necesita ningún argumento.
IsNull	Devuelve aquellos registros en los que el valor de la propiedad dada sea nula

Resumen de métodos dinámicos de GORM

La tabla que se muestra es un resumen de los métodos dinámicos de los que dispone GORM para realizar la búsqueda de registros.

Método	Descripción
`count()`	Devuelve el número total de registros de una determinada clase de dominio
`countBy()`	Devuelve el número total de registros de una determinada clase de dominio que cumplan una serie de requisitos encadenados tras el nombre del método
`findBy()`	Devuelve el primer registro encontrado de una determinada clase de dominio que cumpla una serie de requisitos encadenados tras el nombre del método
`findWhere()`	Devuelve el primer registro encontrado de una determinada clase de dominio que cumpla una serie de requisitos pasados por parámetro en forma de mapa
`findAllBy()`	Devuelve todos los registros encontrados de una determinada clase de dominio que cumplan una serie de requisitos encadenados tras el nombre del método
`findAllWhere()`	Devuelve todos los registros encontrados de una determinada clase de dominio que cumplan una serie de requisitos pasados por parámetro

	en forma de mapa
`getAll()`	Devuelve todos los registros de una determinada clase de dominio cuyo identificador coincida con cualquier de los pasados parámetro en forma de lista
`list()`	Devuelve todos los registros de una determinada clase de dominio. Acepta los parámetros `max`, `offset`, `sort` y `order`
`listOrderBy()`	Devuelve todos los registros de una determinada clase de dominio ordenador por un criterio. Acepta los parámetros `max`, `offset` y `order`

1.2. Consultas HQL de Hibernate

Otra forma de realizar consultas sobre la base de datos es mediante el lenguaje de consulta propio de Hibernate, HQL (Hibernate Query Language). Con los métodos dinámicos de GORM vimos que teníamos la posibilidad de ejecutar llamadas a métodos para obtener un registro o un conjunto. Con HQL vamos a tener también esta capacidad con los métodos `find()` y `findAll()` más uno nuevo llamado `executeQuery()`.

Con el método `find()` vamos a poder obtener el primer registro devuelto de la consulta HQL pasada por parámetro. El siguiente ejemplo, devuelve el primer usuario administrador que encuentre en la clase de dominio *Usuario*.

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

Pero lo habitual no es simplemente obtener el primer registro sino que normalmente necesitaremos obtener todos los registros que cumplan una serie de condiciones para posteriormente, actuar en consecuencia. Al igual que con los métodos dinámicos de GORM, HQL dispone también del método `findAll()` al cual debemos pasarle por parámetro la sentencia HQL correspondiente. En el siguiente ejemplo, vamos a ver tres formas diferentes de obtener los usuarios de tipo *socio* de nuestra aplicación.

```
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",
[tipo:"socio"])
assert hqlsentence4.size()==2
```

En la primera de ellas, la sentencia HQL se pasa como parámetro al método `findAll()`. También es posible pasar parámetros a la sentencia HQL tal y como se ha hecho en la

segunda forma. Utilizando el caracter *?* le especificamos que es valor se pasará en forma de parámetro posicional, es decir, el primer caracter *?* encontrado en la sentencia HQL se referirá al primer parámetro pasado, el segundo *?* al segundo parámetro y así sucesivamente.

Otra forma posible de implementar sentencias HQL es pasando un mapa con las variables introducidas en la sentencia HQL. Para introducir una variable en una sentencia HQL basta con escribir el carácter *:* seguido del nombre de la variable que deseemos crear. En el ejemplo hemos creado la variable *tipo*, que posteriormente se rellenará en el mapa pasado como parámetro.

De igual forma que veíamos anteriormente con los métodos `list()` y `listOrderBy()`, HQL permite utilizar los parámetros `max`, `offset`, `sort` y `order` para afinar aún más la búsqueda.

Por último, el método `executeQuery()` es algo diferente a los métodos `find()` y `findAll()`. En estos dos métodos se devolvían todas las columnas de la tabla en cuestión, cosa que no siempre puede ser necesaria. El método `executeQuery()` permite seleccionar que columnas vamos a devolver. El siguiente ejemplo devuelve sólo el *nombre* y *apellidos* de los administradores del sistema.

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

1.3. Consultas Criteria de Hibernate

El último método para realizar consultas a la base de datos se refiere a la utilización de *Criteria*, un API de Hibernate diseñado específicamente para la realización de consultas complejas. En Grails, Criteria está basado en un *Builder de Groovy* y el código mostrado a continuación es un ejemplo de utilización de éste. En este ejemplo vamos a ver como podríamos obtener un listado de todas las operaciones de tipo *préstamo* realizadas los últimos 10 días.

```
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
}
```

Criteria dispone de una serie de *criterios* disponibles para ser utilizados en este tipo de consultas. En el ejemplo hemos utilizado `between`, `eq`, `maxResults` y `order`, pero existen muchos más que vamos a ver en la siguiente tabla.

Criterio	Descripción	Ejemplo
between	Cuando el valor de la	`between("fechaInicio",new`

	propiedad se encuentra entre dos valores	`Date()-10, new Date())`
eq	Cuando el valor de una propiedad es igual al valor pasado por parámetro	`eq("tipo","prestamo")`
eqProperty	Cuando dos propiedades tienen el mismo valor	`eqProperty("fechaInicio","fechaFin")`
gt	Cuando el valor de una propiedad es mayor que el valor pasado por parámetro	`gt("fechaInicio",new Date()-5)`
gtProperty	Cuando el valor de una propiedad es mayor que el valor de la propiedad pasada por parámetro	`gtProperty("fechaInicio","fechaFin")`
ge	Cuando el valor de una propiedad es mayor o igual que el valor pasado por parámetro	`ge("fechaInicio", new Date()-5)`
geProperty	Cuando el valor de una propiedad es mayor o igual que el valor de la propiedad pasada por parámetro	`geProperty("fechaInicio", "fechaFin")`
idEq	Cuando el identificador de un objeto es igual al valor pasado por parámetro	`idEq(1)`
ilike	La sentencia SQL like, pero en modo *case insensitive*	`ilike("nombre", "fran%")`
in	Cuando el valor de la propiedad es cualquiera de los valores pasados por parámetro. In es una palabra reservada en Groovy, así que debemos utilizar las comillas simples	`'in'("edad",[18..65])`
isEmpty	Cuando la propiedad se encuentra vacía	`isEmpty("descripcion")`
isNotEmpty	Cuando la propiedad no se encuentra vacía	`isNotEmpty("descripcion")`
isNull	Cuando el valor de la propiedad es `null`	`isNull("descripcion")`
isNotNull	Cuando el valor de la propiedad no es `null`	`isNotNull("descripcion")`

lt	Cuando el valor de una propiedad es menor que el valor pasado por parámetro	`lt("fechaInicio",new Date()-5)`
ltProperty	Cuando el valor de una propiedad es menor que el valor de la propiedad pasada por parámetro	`ltProperty("fechaInicio","fechaFi`
le	Cuando el valor de una propiedad es menor o igual que el valor pasado por parámetro	`le("fechaInicio",new Date()-5)`
leProperty	Cuando el valor de una propiedad es menor o igual que el valor de la propiedad pasada por parámetro	`leProperty("fechaInicio","fechaFi`
like	La sentencia SQL like	`like("nombre","Fran%")`
ne	Cuando el valor de una propiedad no es igual al valor pasado por parámetro	`ne("tipo","prestamo")`
neProperty	Cuando el valor de una propiedad no es igual al valor de la propiedad pasada por parámetro	`neProperty("fechaInicio","fechaFi`
order	Ordena los resultados por la propiedad pasada por parámetro y en el orden especificado (asc o desc)	`order("nombre","asc")`
rlike	Similar a la sentencia SQL like, pero utilizando expresiones regulares. Sólo está disponible para Oracle y MySQL	`rlike("nombre",/Fran.+/)`
sizeEq	Cuando el tamaño del valor de una determinada propiedad es igual al pasado por parámetro	`sizeEq("nombre",10)`
sizeGt	Cuando el tamaño del valor de una determinada propiedad es mayor al pasado por parámetro	`sizeGt("nombre",10)`
sizeGe	Cuando el tamaño del valor de una determinada propiedad es mayor o igual al pasado por parámetro	`sizeGe("nombre",10)`
sizeLt	Cuando el tamaño del valor de una determinada propiedad es	`sizeLt("nombre",10)`

	menor al pasado por parámetro	
sizeLe	Cuando el tamaño del valor de una determinada propiedad es menor o igual al pasado por parámetro	`sizeLe("nombre",10)`
sizeNe	Cuando el tamaño del valor de una determinada propiedad no es igual al pasado por parámetro	`sizeNe("nombre",10)`

Los criterios de búsqueda se pueden incluso agrupar en bloques lógicos. Estos bloques lógicos serán del tipo *AND*, *OR* y *NOT*. Veamos algunos ejemplos.

```
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")
}
```

Otro aspecto que hace muy interesante las consultas de tipo Criteria, es la forma que tiene de buscar la información de clases de dominio relacionadas o asociadas. Un ejemplo podría ser el siguiente. Deseamos saber cuantas operaciones hay asociadas a usuarios de tipo socio. El siguiente código haría esto mismo.

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

Simplemente, hemos agrupado dentro del bloque *usuario* las condiciones que queremos que cumplan los usuarios relacionados con las operaciones.

Por último, Criteria también añade la posibilidad de utilizar operaciones agrupadas del estilo *distinct*, *min* o *max* con lo que se conoce como *proyecciones*. Si quisiéramos conocer cuantos usuarios distintos tienen operaciones asignadas, podríamos realizar la siguiente consulta Criteria.

```
def c3 = Operacion.createCriteria()

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

Las posibles métodos que tenemos para agrupar resultados son los mostrados en la siguiente tabla.

Nombre	Descripción	Ejemplo
property	Devuelve la propiedad dada en los resultados	`property("fechaInicio")`
distinct	Devuelve los resultados utilizando una colección de nombres de propiedades	`distinct("fechaInicio", "fechaFin")`
avg	Devuelve la media de los valores de la propiedad dada	`avg("edad")`
count	Devuelve el número de valores de la propiedad dada	`count("nombre")`
countDistinct	Devuelve el número de valores distintos de la propiedad dada	`countDistinct("tipo")`
groupProperty	Agrupa los resultados por la propiedad pasada por parámetro	`groupProperty("usuario")`
max	Devuelve el valor máximo de la propiedad dada	`max("edad")`
min	Devuelve el valor mínimo de la propiedad dada	`min("edad")`
sum	Devuelve la suma de los valores de la propiedad dada	`sum("ingresos")`
rowCount	Devuelve el número de filas devueltas	`rowCount()`

2. Servicios

En cualquier aplicación que utilice el patrón Modelo Vista Controlador, la capa de servicios debe ser la responsable de la lógica de negocio de nuestra aplicación. Si conseguimos esto, nuestra aplicación será fácil de mantener y evolucionará de forma sencilla y eficiente.

Cuando implementamos una aplicación siguiendo el patrón MVC, el principal error que se comete es acumular demasiado código en la capa de control (controladores). Si en lugar de hacer esto, implementásemos servicios encargados de esta tarea, nuestra aplicación sería más fácil de mantener. Pero, ¿qué son los servicios en Grails?.

Siguiendo el paradigma de *convención sobre configuración*, los servicios no son más que clases cuyo nombre termina en *Service* y que se ubica en el directorio *grails-app/services*.

En la aplicación ejemplo que estamos desarrollando como ejemplo, vamos a crear un servicio para dar de alta nuevos usuarios. Para ello, en primer lugar debemos crear el esqueleto del servicio con el comando `grails create-service usuario`. Este comando, además del servicio, también creará un test unitario. La clase de servicio que hemos creado tiene la siguiente estructura

```
package biblioteca

class UsuarioService {

    static transactional = true

    def serviceMethod() {

    }
}
```

Mientras que el contenido del test sería el siguiente

```
package biblioteca

import grails.test.*

class UsuarioServiceTests extends GrailsUnitTestCase {
    protected void setUp() {
        super.setUp()
    }

    protected void tearDown() {
        super.tearDown()
    }

    void testSomething() {

    }
}
```

Pero centrémonos en la clase de servicio *UsuarioService*. En primer lugar, vemos una variable de tipo *booleano* que indica si el servicio en cuestión es *transaccional* o no. Si declaramos el servicio como transaccional, estamos indicando que en caso de que se produzca cualquier error o excepción, las operaciones realizadas serán desechadas.

Si lo que necesitamos es definir la transaccionalidad a nivel de método, podemos conseguirlo utilizando la anotación `@Transactional`.

```
import org.springframework.transaction.annotation.*

class LibroService{
        @Transactional(readOnly = true)
        def listLibros() { Libro.list()}

        @Transactional def updateLibro(){
                //...
        }
}
```

Puedes encontrar más información sobre el uso de la transaccionalidad en la documentación de Spring.

A continuación, vamos a sustituir el método de ejemplo del servicio `serviceMethod()` por uno propio que cree un nuevo usuario en nuestra aplicación. El método recibirá un mapa con todos los parámetros necesarios para la creación del usuario y devolverá una instancia de la clase *Usuario*.

```
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
}
```

En primer lugar, creamos una nueva instancia de la clase *Usuario* a partir de los parámetros recibidos. A continuación se validan las restricciones correspondientes con la nueva instancia con el método `validate()` y en caso de que no haya ningún problema, se almacena la información del nuevo usuario en la base de datos. Si se produce algún error, la propiedad *errors* del objeto *u* contendrá los errores producidos y se devolverá al controlador los datos del nuevo usuario (que no se ha almacenado en la base de datos) con todos los errores para que el controlador decida que hacer.

El siguiente paso, será modificar el controlador de la clase de dominio *Usuario* para que en lugar de implementar él mismo la lógica de negocio para dar de alta un usuario, le ceda esta parte al servicio recién creado.

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

Al inicio del controlador debemos crear también la variable `def usuarioService` para poder utilizar sus servicios. La convención es que esta variable debe llamarse igual que el servicio salvo que la primera letra será minúscula. De esta forma, una instancia del servicio correspondiente se inyectará convenientemente en nuestro controlador, con lo que no debemos preocuparnos por su ciclo de vida.

Por defecto, todos los servicios en Grails son de tipo *singleton*, es decir, sólo existe una instancia de la clase que se inyecta en todos los artefactos que declaren la variable correspondiente. Esto puede servirnos para la mayoría de las situaciones, pero tiene un inconveniente y es que no es posible guardar información *privada* de una petición en el propio servicio puesto que todos los controladores verían la misma instancia y por lo

tanto, el mismo valor. Para solucionar esto, podemos declarar una variable `scope` con cualquiera de los siguiente valores:

- *prototype*: cada vez que se inyecta el servicio en otro artefacto se crea una nueva instancia
- *request*: se crea una nueva instancia para cada solicitud HTTP
- *flash*: cada instancia estará accesible para la solicitud HTTP actual y para la siguiente
- *flow*: cuando declaramos el servicio en un web flow, se creará una instancia nueva en cada flujo
- *conversation*: cuando declaramos el servicio en un web flow, la instancia será visible para el flujo actual y todos sus sub-flujos (conversación)
- *session*: se creará una instancia nueva del servicio para cada sesión de usuario
- *singleton*: sólo existe una instancia del servicio

Por lo tanto, si queremos modificar el ámbito del servicio para que cada sesión de usuario tenga su propia instancia del servicio, debemos declararlos así:

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

2.1. Servicio de mensajería

Un servicio típico de cualquier aplicación web suele ser el que se refiere al envío de correos electrónicos. En nuestra aplicación, vamos a crear un servicio que sea capaz de enviar emails a los usuarios de nuestro sistema. Este servicio podrá ser utilizado para por ejemplo, el envío de avisos a los usuarios que deben devolver libros puesto que el préstamo de los mismos ha caducado o incluso para avisarles de que el libro que habían solicitado está disponible.

En Grails existe un plugin llamado Mail que facilita muchísimo la labor de envío de correos electrónicos, así que lo primero que tenemos que hacer es instalarlo con el comando `grails install-plugin mail`.

La configuración de este plugin es relativamente sencilla y sólo es necesario añadir un nuevo parámetro en el archivo *grails-app/conf/Config.groovy* para indicarle el servidor SMTP que vamos a utilizar en el envío de los emails. En nuestro caso, podemos hacer uso del servidor de correo de la Universidad de Alicante, con lo que añadiremos:

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

Si necesitamos configurar otro gestor de correo diferente podemos acceder a la documentación del plugin donde hay varios ejemplos de configuraciones.

Una vez instalado y configurado el plugin, vamos a crear un nuevo servicio que hará de intermediario entre los controladores y el plugin. Al nuevo servicio lo vamos a llamar *Notificador* y para crearlo ejecutamos el comando `grails create-service`

`Notificador`.

Este servicio por el momento sólo tendrá un método que será el encargado de enviar los emails y lo llamaremos `mandarMails()`. Además, en el servicio necesitamos definir la llamada al servicio instalado con el plugin *Mail*.

```
package biblioteca

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"
        }
    }
}
```

Ahora que ya tenemos creado el servicio, vamos a probarlo creando un nuevo método en el controlador de la clase *Usuario* para enviar un mail de prueba. El nuevo método se llamará `enviarEmails()` y enviará un email de prueba a nuestra propia dirección de correo electrónico. Posteriormente, mostrará por pantalla un texto indicando que el email se han enviado correctamente.

No olvidas inyectar el servicio *notificador* en el controlador de la clase *Usuario* indicando al inicio de la clase el código `def notificadorService`.

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

Si ahora accedemos a la dirección http://localhost:8080/biblioteca/usuario/enviarEmails comprobaremos como el correo electrónico se ha enviado correctamente.